Introduction to computing with finite difference methods

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Jul 24, 2015

Contents

List of Exercises, Problems, and Projects

Finite difference methods for partial differential equations (PDEs) employ a range of concepts and tools that can be introduced and illustrated by way simple ordinary differential equation (ODE) examples. The aim of the present document is to lay a foundation for understanding numerical methods for PDEs by first meeting the fundamental ideas in a simpler ODE setting. With the ODEs, the mathematical problems are kept as simple as possible (but no simpler!), allowing full focus on the understanding of key concepts and tools. The choice of ODE topics to be covered here is thus solely determined by what carries over to the world of numerical solution methods for PDEs.

Theory and practice are primarily illustrated by solving the very simple ODE $u' = -au$, $u(0) = I$, where $a > 0$ is a constant, but we also address the more general model problem $u' = -a(t)u + b(t)$ and the completely general, nonlinear problem $u' = f(u, t)$. The following list of topics will be elaborated on.

- How to think when constructing finite difference methods, with special focus on the Forward Euler, Backward Euler, and Crank-Nicolson (midpoint) schemes
- How to formulate a computational algorithm and translate it into Python code
- How to make curve plots of the solutions
- How to compute numerical errors
- How to compute convergence rates
- How to test that an implementation is correct (verification) and how to automate tests through *test functions* and *unit testing*
- How to work with Python concepts such as arrays, lists, dictionaries, lambda functions, functions in functions (closures)
- How to perform array computing and understand the difference from scalar computing
- How to uncover numerical artifacts in the computed solution
- How to analyze the numerical schemes mathematically to understand why artifacts occur
- How to derive mathematical expressions for various measures of the error in numerical methods, frequently by using the `sympy` software for symbolic computations
- How to understand concepts such as finite difference operators, mesh (grid), mesh functions, stability, truncation error, consistency, and convergence
- How to solve the general nonlinear ODE $u' = f(u, t)$, which is either a scalar ODE or a system of ODEs (i.e., u and f can either be a function or a vector of functions)

- How to access professional packages for solving ODEs
- How the model equation $u' = -au$ arises in a wide range of phenomena in physics, biology, and finance

The exposition in a nutshell.

Everything we cover is put into a practical, hands-on context. All mathematics is translated into working computing codes, and all the mathematical theory of finite difference methods presented here is motivated from a strong need to understand why we occasionally obtain strange results from the programs. Two fundamental questions saturate the text:

- How do we solve a differential equation problem and produce numbers?
- How do we know that the answer is correct?

Besides answering these two questions, you will learn a lot about mathematical modeling in general and the interplay between physics, mathematics, numerical methods, and computer science.

1 Finite difference methods

Contents of this document.

We explain the basic ideas of finite difference methods primarily via the simple ordinary differential equation $u' = -au$. Emphasis is put on the reasoning behind problem discretizing and introduction of key concepts such as mesh, mesh function, finite difference approximations, averaging in a mesh, derivation of algorithms, and discrete operator notation.

1.1 A basic model for exponential decay

Our model problem is perhaps the simplest ordinary differential equation (ODE):

$$u'(t) = -au(t)\,.$$

Here, $u(t)$ is a scalar function of time t, a is a constant, and $u'(t)$ means differentiation with respect to t. This type of equation arises in a number of widely different phenomena where some quantity u undergoes exponential reduction. Examples include radioactive decay, population decay, investment decay, cooling of an object, pressure decay in the atmosphere, and retarded motion in fluids (for some of these models, a can be negative as well), see Section 9

for details and motivation. We have chosen this particular ODE not only because its applications are relevant, but even more because studying numerical solution methods for this particular ODE gives important insight that can be reused in far more complicated settings, in particular when solving diffusion-type partial differential equations.

The exact solution. Although our interest is in *approximate* numerical solutions of $u' = -au$, it is convenient to know the exact analytical solution of the problem so we can compute the error in numerical approximation. The analytical solution of this ODE is found by separation of variables, which results in

$$u(t) = Ce^{-at},$$

for any arbitrary constant C. To obtain a unique solution, we need a condition to fix the value of C. This condition is known as the *initial condition* and stated as $u(0) = I$. That is, we know that the value of u is I when the process starts at $t = 0$. With this knowledge, the exact solution becomes $u(t) = Ie^{-at}$. The initial condition is also crucial for numerical methods: without it, we can never start the numerical algorithms!

Complete problem formulation. The ODE needs an initial condition, and we must also specify a time interval for the solution: $t \in (0, T]$. The point $t = 0$ is not included since we know that $u(0) = I$ and assume that the equation governs u for $t > 0$. Let us now summarize the information that is required to state the complete problem formulation: find $u(t)$ such that

$$u' = -au, \ t \in (0, T], \quad u(0) = I\,. \tag{1}$$

This is known as a *continuous problem* because the parameter t varies continuously from 0 to T. For each t we have a corresponding $u(t)$. There are hence infinitely many values of t and $u(t)$. The purpose of a numerical method is to formulate a corresponding *discrete* problem whose solution is characterized by a finite number of values, which can be computed in a finite number of steps on a computer. Typically, we choose a finite set of time values $t_0, t_1, \ldots, t_{N_t}$, and create algorithms that generate the corresponding u values $u_0, u_1, \ldots, u_{N_t}$.

1.2 The Forward Euler scheme

Solving an ODE like (1) by a finite difference method consists of the following four steps:

1. discretizing the domain,
2. requiring fulfillment of the equation at discrete time points,
3. replacing derivatives by finite differences,

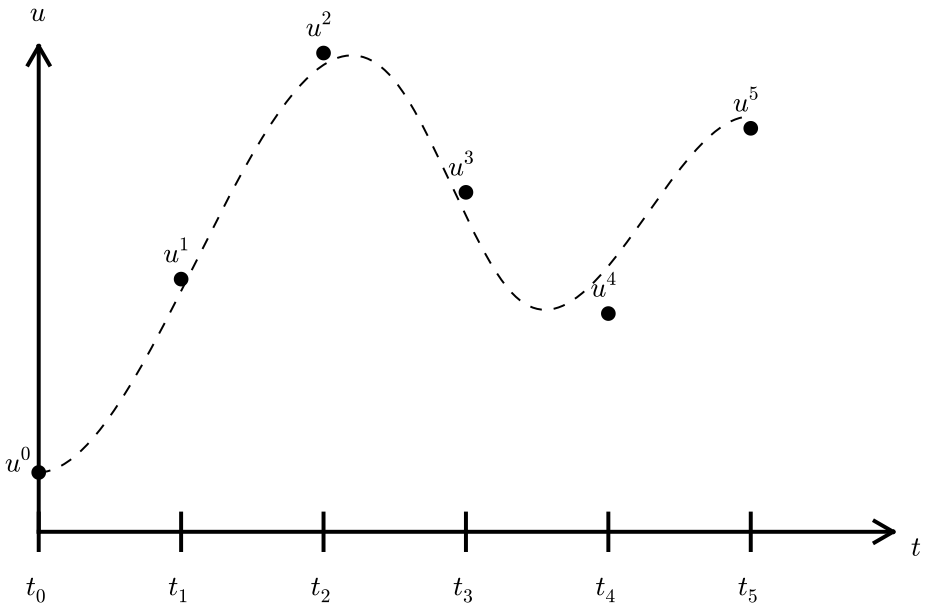


Figure 1: Time mesh with discrete solution values at points and a dashed line indicating the true solution.

4. formulating a recursive algorithm.

Step 1: Discretizing the domain. The time domain $[0, T]$ is represented by a finite number of $N_t + 1$ points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{2}$$

The collection of points $t_0, t_1, \ldots, t_{N_t}$ constitutes a *mesh* or *grid*. Often the mesh points will be uniformly spaced in the domain $[0, T]$, which means that the spacing $t_{n+1} - t_n$ is the same for all n. This spacing is often denoted by Δt, which means that $t_n = n\Delta t$.

We want the solution u at the mesh points: $u(t_n)$, $n = 0, 1, \ldots, N_t$. A notational short-form for $u(t_n)$, which will be used extensively, is u^n. More precisely, we let u^n be the *numerical approximation* to the exact solution $u(t_n)$ at $t = t_n$.

When we need to clearly distinguish between the numerical and exact solution, we often place a subscript e on the exact solution, as in $u_{\mathrm{e}}(t_n)$. Figure 1 shows the t_n and u^n points for $n = 0, 1, \ldots, N_t = 7$ as well as $u_{\mathrm{e}}(t)$ as the dashed line.

We say that the numerical approximation, i.e., the collection of u^n values for $n = 0, \ldots, N_t$, constitutes a *mesh function*. A "normal" continuous function is a curve defined for all real t values in $[0, T]$, but a mesh function is only

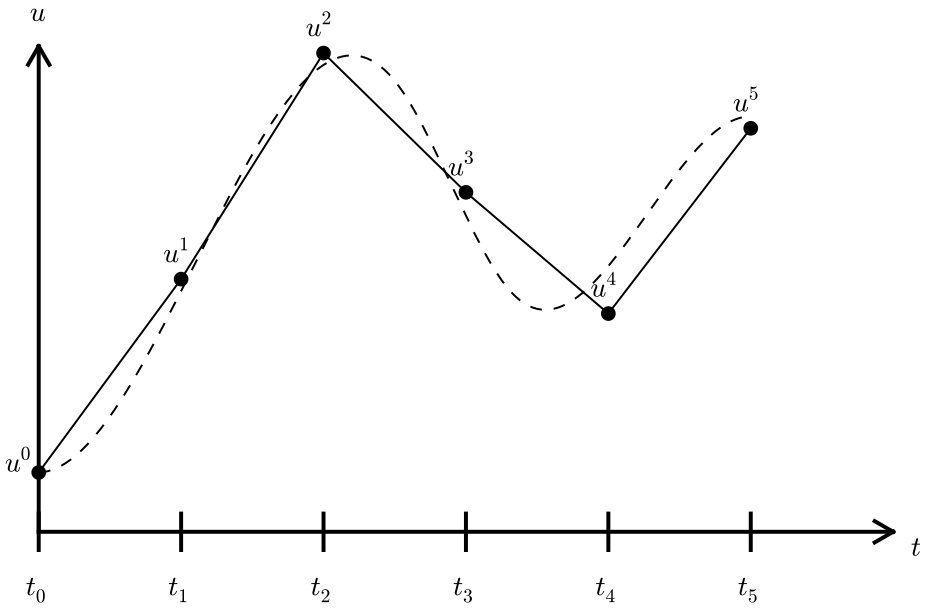


Figure 2: Linear interpolation between the discrete solution values (dashed curve is exact solution).

defined at discrete points in time. If you want to compute the mesh function *between* the mesh points, where it is not defined, an *interpolation method* must be used. Usually, linear interpolation, i.e., drawing a straight line between the mesh function values, see Figure 1, suffices. To compute the solution for some $t \in [t_n, t_{n+1}]$, we use the linear interpolation formula

$$u(t) \approx u^n + \frac{u^{n+1} - u^n}{t_{n+1} - t_n}(t - t_n)\,. \tag{3}$$

Notice.

The goal of a numerical solution method for ODEs is to compute the mesh function by solving a finite set of *algebraic equations* derived from the original ODE problem.

Step 2: Fulfilling the equation at discrete time points. The ODE is supposed to hold for all $t \in (0, T]$, i.e., at an infinite number of points. Now we relax that requirement and require that the ODE is fulfilled at a finite set of discrete points in time. The mesh points $t_0, t_1, \ldots, t_{N_t}$ are a natural (but not

the only) choice of points. The original ODE is then reduced to the following equations:

$$u'(t_n) = -au(t_n), \quad n = 0, \ldots, N_t, \quad u(0) = I\,. \tag{4}$$

Even though the original ODE is not stated to be valid at $t = 0$, it is valid as close to $t = 0$ as we like, and it turns out that it is useful for construction of numerical methods to have (4) valid for $n = 0$. The next two steps show that we need (4) for $n = 0$.

Step 3: Replacing derivatives by finite differences. The next and most essential step of the method is to replace the derivative u' by a finite difference approximation. Let us first try a *forward* difference approximation (see Figure 3),

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{5}$$

The name forward relates to the fact that we use a value forward in time, u^{n+1}, together with the value u^n at the point t_n, where we seek the derivative, to approximate $u'(t_n)$. Inserting this approximation in (4) results in

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^n, \quad n = 0, 1, \ldots, N_t - 1\,. \tag{6}$$

Note that if we want to compute the solution up to time level N_t, we only need (4) to hold for $n = 0, \ldots, N_t - 1$ since (6) for $n = N_t - 1$ creates an equation for the final value u^{N_t}.

Also note that we use the approximation symbol $\approx$ in (5), but not in (6). Instead, we view (6) as an equation that is not mathematically equivalent to (5), but represents an approximation to the equation (5).

Equation (6) is the discrete counterpart to the original ODE problem (1), and often referred to as a *finite difference scheme* or more generally as the *discrete equations* of the problem. The fundamental feature of these equations is that they are *algebraic* and can hence be straightforwardly solved to produce the mesh function, i.e., the approximate values of u at the mesh points: u^n, $n = 1, 2, \ldots, N_t$.

Step 4: Formulating a recursive algorithm. The final step is to identify the computational algorithm to be implemented in a program. The key observation here is to realize that (6) can be used to compute u^{n+1} if u^n is known. Starting with $n = 0$, u^0 is known since $u^0 = u(0) = I$, and (6) gives an equation for u^1. Knowing u^1, u^2 can be found from (6). In general, u^n in (6) can be assumed known, and then we can easily solve for the unknown u^{n+1}:

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n\,. \tag{7}$$

We shall refer to (7) as the Forward Euler (FE) scheme for our model problem. From a mathematical point of view, equations of the form (7) are known as

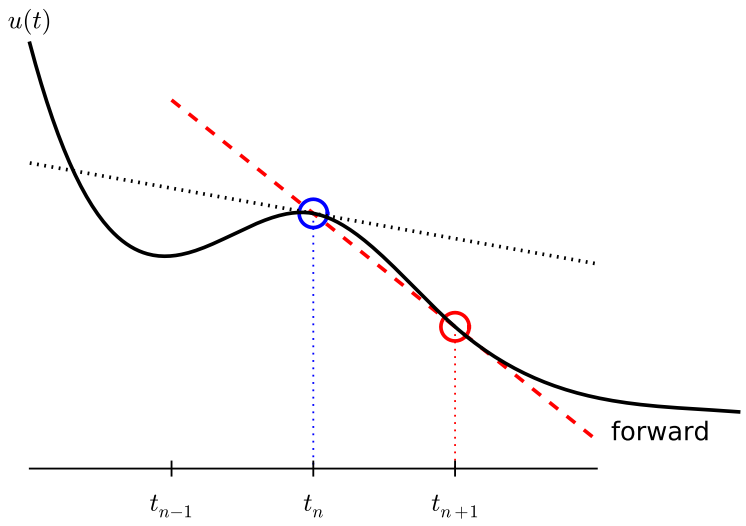
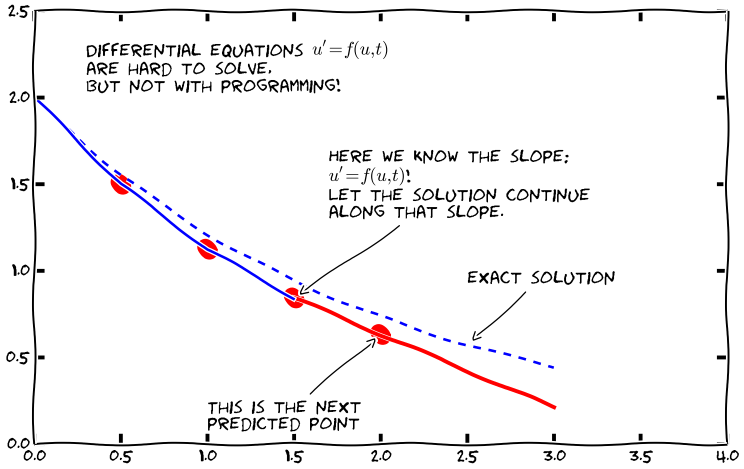


Figure 3: Illustration of a forward difference.

difference equations since they express how differences in the dependent variable, here u, evolve with n. In our case, the differences in u are given by $u^{n+1} - u^n = -a(t_{n+1} - t_n)u^n$. The finite difference method can be viewed as a method for turning a differential equation into an algebraic difference equation that can be easily solved by repeated use of a formula like (7).

Interpretation. There is a very intuitive interpretation of the FE scheme, illustrated in the sketch below. We have computed some point values on the solution curve (small red disks), and the question is how we reason about the next point. Since we know u and t at the most recently computed point, the differential equation gives us the *slope* of the solution curve: $u' = -au$. We can draw this slope as a red line and continue the solution curve along that slope. As soon as we have chosen the next point on this line, we have a new t and u value and compute a new slope and continue the process.



Computing with the recursive formula. Mathematical computation with (7) is straightforward:

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= u^0 - a(t_1 - t_0)u^0 = I(1 - a(t_1 - t_0)),\\
u_2 &= u^1 - a(t_2 - t_1)u^1 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1)),\\
u^3 &= u^2 - a(t_3 - t_2)u^2 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1))(1 - a(t_3 - t_2)),
\end{aligned}
$$

and so on until we reach u^{N_t}. Very often, $t_{n+1} - t_n$ is constant for all n, so we can introduce the common symbol $\Delta t = t_{n+1} - t_n$, $n = 0, 1, \ldots, N_t - 1$. Using a constant mesh spacing Δt in the above calculations gives

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= I(1 - a\Delta t),\\
u_2 &= I(1 - a\Delta t)^2,\\
u^3 &= I(1 - a\Delta t)^3,\\
&\vdots\\
u^{N_t} &= I(1 - a\Delta t)^{N_t}\,.
\end{aligned}
$$

This means that we have found a closed formula for u^n, and there is no need to let a computer generate the sequence $u^1, u^2, u^3, \ldots$. However, finding such a formula for u^n is possible only for a few very simple problems, so in general finite difference equations must be solved on a computer.

As the next sections will show, the scheme (7) is just one out of many alternative finite difference (and other) methods for the model problem (1).

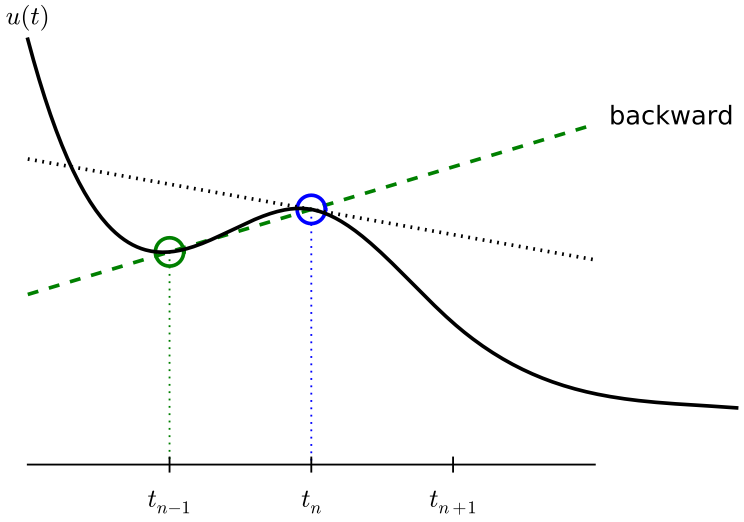


Figure 4: Illustration of a backward difference.

1.3 The Backward Euler scheme

There are several choices of difference approximations in step 3 of the finite difference method as presented in the previous section. Another alternative is

$$u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}}\,. \tag{8}$$

Since this difference is based on going backward in time (t_{n-1}) for information, it is known as a *backward* difference, also called Backward Euler difference. Figure 4 explains the idea.

Inserting (8) in (4) yields the Backward Euler (BE) scheme:

$$\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n, \quad n = 1, \ldots, N_t\,. \tag{9}$$

We assume, as explained under step 4 in Section 1.2, that we have computed $u^0, u^1, \ldots, u^{n-1}$ such that (9) can be used to compute u^n. Note that (9) needs n to start at 1 (then it involves u^0, but no u^{-1}) and end at N_t.

For direct similarity with the formula for the Forward Euler scheme (7) we replace n by $n+1$ in (9) and solve for the unknown value u^{n+1}:

$$u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)}u^n, \quad n = 0, \ldots, N_t - 1\,. \tag{10}$$

1.4 The Crank-Nicolson scheme

The finite difference approximations (5) and (8) used to derive the schemes (7) and (10), respectively, are both one-sided differences, i.e., we collect information

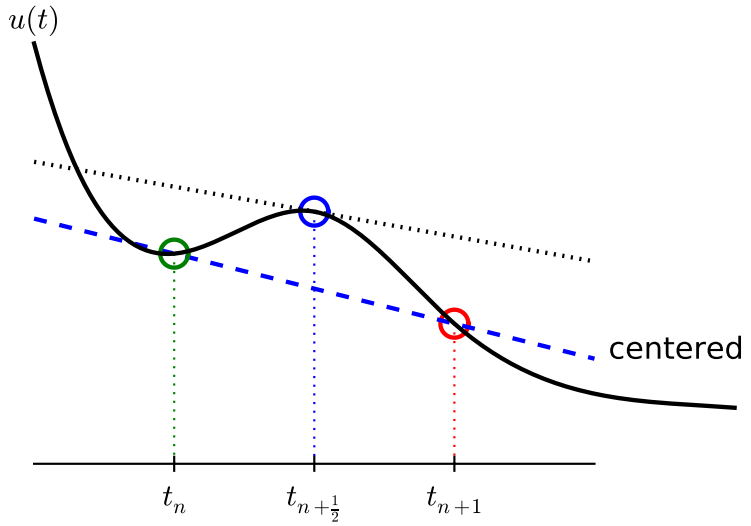


Figure 5: Illustration of a centered difference.

either forward or backward in time when approximating the derivative at a point. Such one-sided differences are known to be less accurate than central (or midpoint) differences, where we use information both forward and backward in time. A natural next step is therefore to construct a central difference approximation that will yield a more accurate numerical solution.

The central difference approximation to the derivative is sought at the point $t_{n+\frac{1}{2}} = \frac{1}{2}(t_n + t_{n+1})$ (or $t_{n+\frac{1}{2}} = (n + \frac{1}{2})\Delta t$ if the mesh spacing is uniform in time). The approximation reads

$$u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{11}$$

Figure 5 sketches the geometric interpretation of such a centered difference. Note that the fraction on the right-hand side is the same as for the Forward Euler approximation (5) and the Backward Euler approximation (8) (with n replaced by $n+1$). The accuracy of this fraction as an approximation to the derivative of u depends on *where* we seek the derivative: in the center of the interval $[t_n, t_{n+1}]$ or at the end points. We shall later see that it is more accurate at the center point.

With the formula (11), where u' is evaluated at $t_{n+\frac{1}{2}}$, it is natural to demand the ODE to be fulfilled at the time points *between* the mesh points:

$$u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}}), \quad n = 0, \ldots, N_t - 1\,. \tag{12}$$

Using (11) in (12) results in the approximate discrete equation

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -au^{n+\frac{1}{2}}, \quad n = 0, \ldots, N_t - 1, \tag{13}$$

where $u^{n+\frac{1}{2}}$ is a short form for the numerical approximation to $u(t_{n+\frac{1}{2}})$.

There is a fundamental problem with the right-hand side of (13): we aim to compute u^n for integer n, which means that $u^{n+\frac{1}{2}}$ is not a quantity computed by our method. The quantity must therefore be expressed by the quantities that we actually produce, i.e., the numerical solution at the mesh points. One possibility is to approximate $u^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighboring mesh points:

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}) \,. \tag{14}$$

Using (14) in (13) results in a new approximate discrete equation

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a\frac{1}{2}(u^n + u^{n+1}) \,. \tag{15}$$

There are three approximation steps leading to this formula: 1) the ODE is only valid at discrete points (between the mesh points), 2) the derivative is approximated by a finite difference, and 3) the value of u between mesh points is approximated by an arithmetic mean value. Despite one more approximation than for the Backward and Forward Euler schemes, the use of a centered difference leads to a more accurate method.

To formulate a recursive algorithm, we assume that u^n is already computed so that u^{n+1} is the unknown, which we can solve for:

$$u^{n+1} = \frac{1 - \frac{1}{2}a(t_{n+1}-t_n)}{1 + \frac{1}{2}a(t_{n+1}-t_n)}u^n \,. \tag{16}$$

The finite difference scheme (16) is often called the Crank-Nicolson (CN) scheme or a midpoint or centered scheme. Note that (16) as well as (7) and (10) apply whether the spacing in the time mesh, $t_{n+1} - t_n$, depends on n or is constant.

1.5 The unifying θ-rule

The Forward Euler, Backward Euler, and Crank-Nicolson schemes can be formulated as one scheme with a varying parameter θ:

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a(\theta u^{n+1} + (1-\theta)u^n) \,. \tag{17}$$

Observe:

- $\theta = 0$ gives the Forward Euler scheme
- $\theta = 1$ gives the Backward Euler scheme, and

- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme.

We shall later, in Section 4, learn the pros and cons of the three alternatives. One may alternatively choose any other value of θ in $[0, 1]$, but this is not so common since the accuracy and stability of the scheme do not improve compared to the values $\theta = 0, 1, \frac{1}{2}$.

As before, u^n is considered known and u^{n+1} unknown, so we solve for the latter:

$$u^{n+1} = \frac{1 - (1-\theta)a(t_{n+1} - t_n)}{1 + \theta a(t_{n+1} - t_n)}\,. \tag{18}$$

This scheme is known as the θ-rule, or alternatively written as the "theta-rule".

Derivation.

We start with replacing u' by the fraction

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n},$$

in the Forward Euler, Backward Euler, and Crank-Nicolson schemes. Then we observe that the difference between the methods concerns which point this fraction approximates the derivative. Or in other words, at which point we sample the ODE. So far this has been the end points or the midpoint of $[t_n, t_{n+1}]$. However, we may choose any point $\tilde{t} \in [t_n, t_{n+1}]$. The difficulty is that evaluating the right-hand side $-au$ at an arbitrary point faces the same problem as in Section 1.4: the point value must be expressed by the discrete u quantities that we compute by the scheme, i.e., u^n and u^{n+1}. Following the averaging idea from Section 1.4, the value of u at an arbitrary point $\tilde{t}$ can be calculated as a *weighted average*, which generalizes the arithmetic mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. The weighted average reads

$$u(\tilde{t}) \approx \theta u^{n+1} + (1-\theta)u^n, \tag{19}$$

where $\theta \in [0, 1]$ is a weighting factor. We can also express $\tilde{t}$ as a similar weighted average

$$\tilde{t} \approx \theta t_{n+1} + (1-\theta)t_n\,. \tag{20}$$

Let now the ODE hold at the point $\tilde{t} \in [t_n, t_{n+1}]$, approximate u' by the fraction $(u^{n+1} - u^n)/(t_{n+1} - t_n)$, and approximate the right-hand side $-au$ by the weighted average (19). The result is (17).

1.6 Constant time step

All schemes up to now have been formulated for a general non-uniform mesh in time: $t_0 < t_1 < \cdots < t_{N_t}$. Non-uniform meshes are highly relevant since one can use many points in regions where u varies rapidly, and fewer points in regions where u is slowly varying. This idea saves the total number of points and therefore makes it faster to compute the mesh function u^n. Non-uniform meshes are used together with *adaptive* methods that are able to adjust the time mesh during the computations (Section 7.11 applies adaptive methods).

However, a uniformly distributed set of mesh points is not only convenient, but also sufficient for many applications. Therefore, it is a very common choice. We shall present the finite difference schemes for a uniform point distribution $t_n = n\Delta t$, where Δt is the constant spacing between the mesh points, also referred to as the *time step*. The resulting formulas look simpler and are more well known.

Summary of schemes for constant time step.

$$u^{n+1} = (1 - a\Delta t)u^n \qquad \text{Forward Euler} \tag{21}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \qquad \text{Backward Euler} \tag{22}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \qquad \text{Crank-Nicolson} \tag{23}$$

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t}u^n \qquad \text{The } \theta - \text{rule} \tag{24}$$

It is not accidental that we focus on presenting the Forward Euler, Backward Euler, and Crank-Nicolson schemes. They complement each other with their different pros and cons, thus providing a useful collection of solution methods for many differential equation problems. The unifying notation of the θ-rule makes it convenient to work with all three methods through just one formula.

Test your understanding.

To check that key concepts are really understood, the reader is encouraged to apply the explained finite difference techniques to a slightly different equation. For this purpose, we recommend you do Exercise 17 now!

1.7 Mathematical derivation of finite difference formulas

The finite difference formulas for approximating the first derivative of a function have so far been somewhat justified through graphical illustrations in Figures 3, 4,

and 5. The task is to approximate the derivative at a point of a curve using only two function values. By drawing a straight line through the points, we have some approximation to the tangent of the curve and use the slope of this line as an approximation to the derivative. The slope can be computed by inspecting the figures.

However, we can alternatively derive the finite difference formulas by pure mathematics. The key tool for this approach is Taylor series, or more precisely, approximation of functions by lower-order Taylor polynomials. Given a function $f(x)$ that is sufficiently smooth (i.e., $f(x)$ has "enough derivatives"), a Taylor polynomial of degree m can be used to approximate the value of the function $f(x)$ if we know the values of f and its first m derivatives at some other point $x = a$. The formula for the Taylor polynomial reads

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \frac{1}{6}f'''(a)(x-a)^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(a)(x-a)^m\,. \tag{25}$$

For a function of time, $f(t)$, related to a mesh with spacing Δt, we often need the Taylor polynomial approximation at $f(t_n \pm \Delta t)$ given f and its derivatives at $t = t_n$. Replacing x by $t_n + \Delta t$ and a by t_n gives

$$f(t_n + \Delta t) \approx f(t_n) + f'(t_n)\Delta t + \frac{1}{2}f''(t_n)\Delta t^2 + \frac{1}{6}f'''(t_n)\Delta t^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^m\,. \tag{26}$$

The forward difference. We can use (26) to find an approximation for $f'(t_n)$ simply by solving with respect to this quantity:

$$f'(t_n) \approx \frac{f(t_n+\Delta t) - f(t_n)}{\Delta t} - \frac{1}{2}f''(t_n)\Delta t - \frac{1}{6}f'''(t_n)\Delta t^2 + \cdots - \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^{m-1}\,. \tag{27}$$

By letting $m \to \infty$, this formula is exact, but that is not so much of practical value. A more interesting observation is that all the power terms in Δt vanish as $\Delta t \to 0$, i.e., the formula

$$f'(t_n) \approx \frac{f(t_n+\Delta t) - f(t_n)}{\Delta t} \tag{28}$$

is exact in the limit $\Delta t \to 0$.

The interesting feature of (27) is that we have a measure of the error in the formula (28): the error is given by the extra terms on the right-hand side of (27). We assume that Δt is a small quantity ($\Delta t \ll 1$). Then $\Delta t^2 \ll \Delta t$,

$\Delta t^3 \ll \Delta t^2$, and so on, which means that the first term is the dominating term. This first term reads $-\frac{1}{2}f''(t_n)\Delta t$ and can be taken as a measure of the error in the Forward Euler formula.

The backward difference. To derive the backward difference, we use the Taylor polynomial approximation at $f(t_n - \Delta t)$:

$$f(t_n - \Delta t) \approx f(t_n) - f'(t_n)\Delta t + \frac{1}{2}f''(t_n)\Delta t^2 - \frac{1}{6}f'''(t_n)\Delta t^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^m \,. \tag{29}$$

Solving with respect to $f'(t_n)$ gives

$$f'(t_n) \approx \frac{f(t_n) - f(t_n - \Delta t)}{\Delta t} + \frac{1}{2}f''(t_n)\Delta t - \frac{1}{6}f'''(t_n)\Delta t^2 + \cdots - \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^{m-1} \,. \tag{30}$$

The term $\frac{1}{2}f''(t_n)\Delta t$ can be taken as a simple measure of the approximation error since it will dominate over the other terms as $\Delta t \to 0$.

The centered difference. The centered difference approximates the derivative at $t_n + \frac{1}{2}\Delta t$. Let us write up the Taylor polynomial approximations to $f(t_n)$ and $f(t_{n+1})$ around $t_n + \frac{1}{2}\Delta t$:

$$f(t_n) \approx f(t_n + \frac{1}{2}\Delta t) - f'(t_n + \frac{1}{2}\Delta t)\frac{1}{2}\Delta t + f''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^2 - f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{31}$$

$$f(t_{n+1}) \approx f(t_n + \frac{1}{2}\Delta t) + f'(t_n + \frac{1}{2}\Delta t)\frac{1}{2}\Delta t + f''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^2 + f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{32}$$

Subtracting the first from the second gives

$$f(t_{n+1}) - f(t_n) = f'(t_n + \frac{1}{2}\Delta t)\Delta t + 2f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{33}$$

Solving with respect to $f'(t_n + \frac{1}{2}\Delta t)$ results in

$$f'(t_n + \frac{1}{2}\Delta t) \approx \frac{f(t_{n+1}) - f(t_n)}{\Delta t} - \frac{1}{4}f'''(t_n + \frac{1}{2}\Delta t)\Delta t^2 + c\cdots \tag{34}$$

This time the error measure goes like $\frac{1}{4}f'''\Delta t^2$, i.e., it is proportional to Δt^2 and not only Δt, which means that the error goes faster to zero as Δt is reduced. This means that the centered difference formula

$$f'(t_n + \frac{1}{2}\Delta t) \approx \frac{f(t_{n+1}) - f(t_n)}{\Delta t} \tag{35}$$

is more accurate than the forward and backward differences for small Δt.

1.8 Compact operator notation for finite differences

Finite difference formulas can be tedious to write and read, especially for differential equations with many terms and many derivatives. To save space and help the reader spot the nature of the difference approximations, we introduce a compact notation. For a function $u(t)$, a forward difference approximation is denoted by the D_t^+ operator and written as

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \quad \left(\approx \frac{d}{dt} u(t_n) \right) . \tag{36}$$

The notation consists of an operator that approximates differentiation with respect to an independent variable, here t. The operator is built of the symbol D, with the independent variable as subscript and a superscript denoting the type of difference. The superscript $^+$ indicates a forward difference. We place square brackets around the operator and the function it operates on and specify the mesh point, where the operator is acting, by a superscript after the closing bracket.

The corresponding operator notation for a centered difference and a backward difference reads

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt} u(t_n), \tag{37}$$

and

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt} u(t_n) . \tag{38}$$

Note that the superscript $^-$ denotes the backward difference, while no superscript implies a central difference.

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n) \tag{39}$$

The superscript t indicates that the average is taken along the time coordinate. The common average $(u^n + u^{n+1})/2$ can now be expressed as $[\overline{u}^t]^{n+\frac{1}{2}}$. (When also spatial coordinates enter the problem, we need the explicit specification of the coordinate after the bar.)

With our compact notation, the Backward Euler finite difference approximation to $u' = -au$ can be written as

$$[D_t^- u]^n = -au^n \,.$$

In difference equations we often place the square brackets around the whole equation, to indicate at which mesh point the equation applies, since each term must be approximated at the same point:

$$[D_t^- u = -au]^n \,. \tag{40}$$

Similarly, the Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n, \tag{41}$$

while the Crank-Nicolson scheme is written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}} \,. \tag{42}$$

Question.

By use of (37) and (39), are you able to write out the expressions in (42) to verify that it is indeed the Crank-Nicolson scheme?

The θ-rule can be specified in operator notation by

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \,. \tag{43}$$

We define a new time difference

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n}, \tag{44}$$

to be applied at the time point $t_{n+\theta} \approx \theta t_n + (1-\theta)t_{n+1}$. This weighted average gives rise to the *weighted averaging operator*

$$[\overline{u}^{t,\theta}]^{n+\theta} = (1-\theta)u^n + \theta u^{n+1} \approx u(t_{n+\theta}), \tag{45}$$

where $\theta \in [0,1]$ as usual. Note that for $\theta = \frac{1}{2}$ we recover the standard centered difference and the standard arithmetic mean. The idea in (43) is to sample the equation at $t_{n+\theta}$, use a non-symmetric difference at that point $[\bar{D}_t u]^{n+\theta}$, and a weighted (non-symmetric) mean value.

An alternative and perhaps clearer notation is

$$[D_t u]^{n+\frac{1}{2}} = \theta[-au]^{n+1} + (1-\theta)[-au]^n \,.$$

Looking at the various examples above and comparing them with the underlying differential equations, we see immediately which difference approximations that have been used and at which point they apply. Therefore, the compact notation effectively communicates the reasoning behind turning a differential equation into a difference equation.

2 Implementation

Goal.

We want to make a computer program for solving

$$u'(t) = -au(t), \quad t \in (0, T], \quad u(0) = I,$$

by finite difference methods. The program should also display the numerical solution as a curve on the screen, preferably together with the exact solution.

All programs referred to in this section are found in the `src/decay` directory (we use the classical Unix term *directory* for what many others nowadays call *folder*).

Mathematical problem. We want to explore the Forward Euler scheme, the Backward Euler, and the Crank-Nicolson schemes applied to our model problem. From an implementational point of view, it is advantageous to implement the θ-rule

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a \Delta t} u^n,$$

since it can generate the three other schemes by various of choices of θ: $\theta = 0$ for Forward Euler, $\theta = 1$ for Backward Euler, and $\theta = 1/2$ for Crank-Nicolson. Given a, $u^0 = I$, T, and Δt, our task is to use the θ-rule to compute $u^1, u^2, \ldots, u^{N_t}$, where $t_{N_t} = N_t \Delta t$, and N_t the closest integer to $T/\Delta t$.

Computer Language: Python. Any programming language can be used to generate the u^{n+1} values from the formula above. However, in this document we shall mainly make use of Python. There several good reasons for this choice:

- Python has a very clean, readable syntax (often known as "executable pseudo-code").
- Python code is very similar to MATLAB code (and MATLAB has a particularly widespread use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to C++, but is much simpler to work with and results in more reliable code.
- Python has a rich set of modules for scientific computing, and its popularity in scientific computing is rapidly growing.
- Python was made for being combined with compiled languages (C, C++, Fortran), so that existing numerical software can be reused, and thereby easing high computational performance with new implementations.

- Python has extensive support for administrative tasks needed when doing large-scale computational investigations.
- Python has extensive support for graphics (visualization, user interfaces, web applications).

Learning Python is easy. Many newcomers to the language will probably learn enough from the forthcoming examples to perform their own computer experiments. The examples start with simple Python code and gradually make use of more powerful constructs as we proceed. Unless it is inconvenient for the problem at hand, our Python code is made as close as possible to MATLAB code for easy transition between the two languages.

The coming programming examples assumes familiarity with variables, for loops, lists, arrays, functions, positional arguments, and keyword (named) arguments. A background in basic MATLAB programming is often enough to understand Python examples. Readers who feel the Python examples are too hard to follow will benefit from reading a tutorial, e.g.,

- The Official Python Tutorial
- Python Tutorial on tutorialspoint.com
- Interactive Python tutorial site
- A Beginner's Python Tutorial

The author also has a comprehensive book [7] that teaches scientific programming with Python from the ground up.

2.1 Making a solver function

We choose to have an array `u` for storing the u^n values, $n = 0, 1, \ldots, N_t$. The algorithmic steps are

1. initialize u^0
2. for $t = t_n$, $n = 1, 2, \ldots, N_t$: compute u_n using the θ-rule formula

An implementation of a numerical algorithm is often referred to as a *solver*. We shall now make a solver for our model problem and realize the solver as a Python function. The function must take the input data I, a, T, Δt, and θ of the problem as arguments and return the solution as arrays `u` and `t` for u^n and t^n, $n = 0, \ldots, N_t$. The solver function used as

```
u, t = solver(I, a, T, dt, theta)
```

One can now easily plot `u` versus `t` to visualize the solution.

Function for computing the numerical solution. The function `solver` may look as follows in Python:

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    Nt = int(T/dt)            # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

The `numpy` library contains a lot of functions for array computing. Most of the function names are similar to what is found in the alternative scientific computing language MATLAB. Here we make use of

- `zeros(Nt+1)` for creating an array of size `Nt+1` and initializing the elements to zero
- `linspace(0, T, Nt+1)` for creating an array with `Nt+1` coordinates uniformly distributed between `0` and `T`

The `for` loop deserves a comment, especially for newcomers to Python. The construction `range(0, Nt, s)` generates all integers from `0` to `Nt` in steps of `s`, *but not including* `Nt`. Omitting `s` means `s=1`. For example, `range(0, 6, 3)` gives `0` and `3`, while `range(0, 6)` generates the list `[0, 1, 2, 3, 4, 5]`. Our loop implies the following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is what we want since `u` has length `Nt+1`. The first index in Python arrays or lists is *always* `0` and the last is then `len(u)-1` (the length of an array `u` is obtained by `len(u)` or `u.size`).

Integer division. The shown implementation of the `solver` may face problems and wrong results if `T`, `a`, `dt`, and `theta` are given as integers (see Exercises 3 and 4). The problem is related to *integer division* in Python (as in Fortran, C, C++, and many other computer languages!): `1/2` becomes `0`, while `1.0/2`, `1/2.0`, or `1.0/2.0` all become `0.5`. So, it is enough that at least the nominator or the denominator is a real number (i.e., a `float` object) to ensure a correct mathematical division. Inserting a conversion `dt = float(dt)` guarantees that `dt` is `float`.

Another problem with computing $N_t = T/\Delta t$ is that we should round N_t to the nearest integer. With `Nt = int(T/dt)` the `int` operation picks the largest integer smaller than `T/dt`. Correct mathematical rounding as known from school is obtained by

```
Nt = int(round(T/dt))
```

The complete version of our improved, safer `solver` function then becomes

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Doc strings. Right below the header line in the `solver` function there is a Python string enclosed in triple double quotes `"""`. The purpose of this string object is to document what the function does and what the arguments are. In this case the necessary documentation do not span more than one line, but with triple double quoted strings the text may span several lines:

```
def solver(I, a, T, dt, theta):
    """
    Solve

        u'(t) = -a*u(t),

    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.

    theta=1 corresponds to the Backward Euler scheme, theta=0
    to the Forward Euler scheme, and theta=0.5 to the Crank-
    Nicolson method.
    """
    ...
```

Such documentation strings appearing right after the header of a function are called *doc strings*. There are tools that can automatically produce nicely formatted documentation by extracting the definition of functions and the contents of doc strings.

It is strongly recommended to equip any function with a doc string, unless the purpose of the function is not obvious. Nevertheless, the forthcoming text deviates from this rule if the function is explained in the text.

Formatting numbers. Having computed the discrete solution `u`, it is natural to look at the numbers:

```
# Write out a table of t and u values:
for i in range(len(t)):
    print t[i], u[i]
```

This compact `print` statement unfortunately gives less readable output because the `t` and `u` values are not aligned in nicely formatted columns. To fix this problem, we recommend to use the *printf format*, supported in most programming languages inherited from C. Another choice is Python's recent *format string syntax*. Both kind of syntax is illustrated below.

Writing `t[i]` and `u[i]` in two nicely formatted columns is done like this with the printf format:

```
print 't=%6.3f u=%g' % (t[i], u[i])
```

The percentage signs signify "slots" in the text where the variables listed at the end of the statement are inserted. For each "slot" one must specify a format for how the variable is going to appear in the string: `f` for float (with 6 decimals), `s` for pure text, `d` for an integer, `g` for a real number written as compactly as possible, `9.3E` for scientific notation with three decimals in a field of width 9 characters (e.g., `-1.351E-2`), or `.2f` for standard decimal notation with two decimals formatted with minimum width. The printf syntax provides a quick way of formatting tabular output of numbers with full control of the layout.

The alternative *format string syntax* looks like

```
print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

As seen, this format allows logical names in the "slots" where `t[i]` and `u[i]` are to be inserted. The "slots" are surrounded by curly braces, and the logical name is followed by a colon and then the printf-like specification of how to format real numbers, integers, or strings.

Running the program. The function and main program shown above must be placed in a file, say with name `decay_v1.py` (`v1` for 1st version of this program). Make sure you write the code with a suitable text editor (Gedit, Emacs, Vim, Notepad++, or similar). The program is run by executing the file this way:

```
Terminal> python decay_v1.py
```

The text `Terminal>` just indicates a prompt in a Unix/Linux or DOS terminal window. After this prompt, which may look different in your terminal window (depending on the terminal application and how it is set up), commands like `python decay_v1.py` can be issued. These commands are interpreted by the operating system.

We strongly recommend to run Python programs within the IPython shell. First start IPython by typing `ipython` in the terminal window. Inside the IPython shell, our program `decay_v1.py` is run by the command `run decay_v1.py`:

```
Terminal> ipython

In [1]: run decay_v1.py
t= 0.000 u=1
t= 0.800 u=0.384615
t= 1.600 u=0.147929
t= 2.400 u=0.0568958
t= 3.200 u=0.021883
t= 4.000 u=0.00841653
t= 4.800 u=0.00323713
t= 5.600 u=0.00124505
t= 6.400 u=0.000478865
t= 7.200 u=0.000184179
t= 8.000 u=7.0838e-05

In [2]:
```

The advantage of running programs in IPython are many: previous commands are easily recalled with the up arrow, `%pdb` turns on a debugger so that variables can be examined if the program aborts (due to a Python exception), output of commands are stored in variables, the computing time spent on a set of statements can be measured, any operating system command can be executed, modules can be loaded automatically and other customizations can be performed when starting IPython – to mention a few of the most useful features.

Although running programs in IPython is strongly recommended, most execution examples in the forthcoming text use the standard Python shell with prompt `>>>` and run programs through a typesetting like

```
Terminal> python programname
```

The reason is that such typesetting makes the text more compact in the vertical direction than showing sessions with IPython syntax.

Plotting the solution. Having the `t` and `u` arrays, the approximate solution `u` is visualized by the intuitive command `plot(t, u)`:

```
from matplotlib.pyplot import *
plot(t, u)
show()
```

It will be illustrative to also plot the exact solution $u_{\mathrm{e}}(t) = Ie^{-at}$ for comparison. We first need to make a Python function for computing the exact solution:

```
def exact_solution(t, I, a):
    return I*exp(-a*t)
```

It is tempting to just do

```
u_e = exact_solution(t, I, a)
plot(t, u, t, u_e)
```

However, this is not exactly what we want: the `plot` function draws straight lines between the discrete points `(t[n], u_e[n])` while $u_{\mathrm{e}}(t)$ varies as an exponential function between the mesh points. The technique for showing the "exact" variation of $u_{\mathrm{e}}(t)$ between the mesh points is to introduce a very fine mesh for $u_{\mathrm{e}}(t)$:

```
t_e = linspace(0, T, 1001)      # fine mesh
u_e = exact_solution(t_e, I, a)
```

We can also plot the curves with different colors and styles, e.g.,

```
plot(t_e, u_e, 'b-',          # blue line for u_e
     t,   u,   'r--o')        # red dashes w/circles
```

With more than one curve in the plot we need to associate each curve with a legend. We also want appropriate names on the axes, a title, and a file containing the plot as an image for inclusion in reports. The Matplotlib package (`matplotlib.pyplot`) contains functions for this purpose. The names of the functions are similar to the plotting functions known from MATLAB. A complete function for creating the comparison plot becomes

```
from matplotlib.pyplot import *

def plot_numerical_and_exact(theta, I, a, T, dt):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(I=I, a=a, T=T, dt=dt, theta=theta)

    t_e = linspace(0, T, 1001)        # fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plot(t,   u,   'r--o',            # red dashes w/circles
         t_e, u_e, 'b-')              # blue line for exact sol.
    legend(['numerical', 'exact'])
    xlabel('t')
    ylabel('u')
    title('theta=%g, dt=%g' % (theta, dt))
    savefig('plot_%s_%g.png' % (theta, dt))

plot_numerical_and_exact(I=1, a=2, T=8, dt=0.8, theta=1)
show()
```

Note that `savefig` here creates a PNG file whose name includes the values of θ and Δt so that we can easily distinguish files from different runs with θ and Δt.

The complete code is found in the file `decay_v2.py`. The resulting plot is shown in Figure 6. As seen, there is quite some discrepancy between the exact

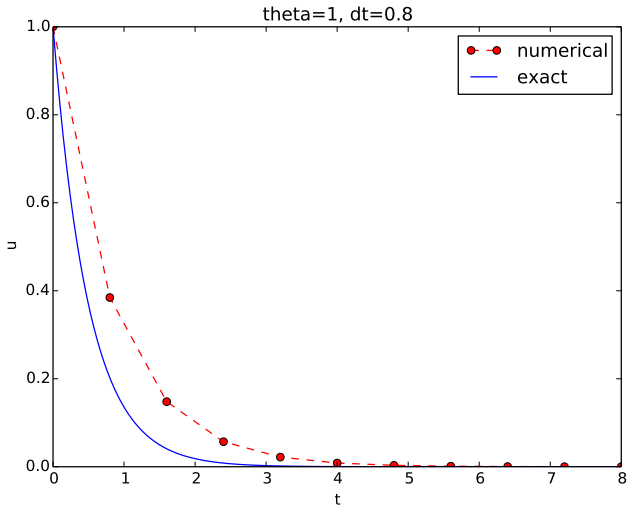


Figure 6: Comparison of numerical and exact solution.

and the numerical solution. Fortunately, the numerical solution approaches the exact one as Δt is reduced.

2.2 Verifying the implementation

It is easy to make mistakes while deriving and implementing numerical algorithms, so we should never believe in the solution before it has been thoroughly verified. The most obvious idea for verification of the computations is to compare the numerical solution with the exact solution, when that exists. However, note that there will always be a discrepancy between these two solutions because of the numerical approximations. We cannot precisely quantify the approximation errors. The challenging question is therefore whether we have the mathematically correct discrepancy or if we have another, maybe small, discrepancy due to both an approximation error *and* an error in the implementation. It is thus impossible to judge whether the program is correct or not by just looking at the graphs in Figure 6.

Verification and validation.

The purpose of *verifying* a program is to bring evidence for the property that there are no errors in the implementation. A related term, *validate* (and *validation*), addresses the question if the ODE model is a good representation of the phenomena we want to simulate. To remember the difference between verification and validation, verification is about *solving*

the equations right, while validation is about *solving the right equations*. We must always perform a verification before it is meaningful to believe in the computations and perform validation (which compares the program results with physical experiments or observations).

To avoid mixing the unavoidable numerical approximation errors and the undesired implementation errors, we should try to make tests where we have some exact computation of the discrete solution or at least parts of it. Examples will show how this can be done.

Running a few algorithmic steps by hand. The simplest approach to produce a correct non-trivial reference solution for the discrete solution u, is to compute a few steps of the algorithm by hand. Then we can compare the hand calculations with numbers produced by the program.

A straightforward approach is to use a calculator and compute u^1, u^2, and u^3. With $I = 0.1$, $\theta = 0.8$, and $\Delta t = 0.8$ we get

$$A \equiv \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t} = 0.298245614035$$

$$\begin{aligned} u^1 &= AI = 0.0298245614035, \\ u^2 &= Au^1 = 0.00889504462912, \\ u^3 &= Au^2 = 0.00265290804728 \end{aligned}$$

Comparison of these manual calculations with the result of the `solver` function is carried out in the function

```
def test_solver_three_steps():
    """Compare three steps with known manual computations."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    u_by_hand = array([I,
                       0.0298245614035,
                       0.00889504462912,
                       0.00265290804728])

    Nt = 3  # number of time steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)

    tol = 1E-15  # tolerance for comparing floats
    diff = abs(u - u_by_hand).max()
    success = diff <= tol
    assert success
```

The `test_solver_three_steps` function follows widely used conventions for *unit testing*. By following such conventions we can at a later stage easily execute a big test suite for our software. That is, after a small modification is made to the program, we can by typing just a short command, run through a large number of tests to check that the modifications do not break any computations. The conventions boil down to three rules:

- The test function name must start with `test_` and the function cannot take any arguments.
- The test must end up in a boolean expression that is `True` if the test was passed and `False` if it failed.
- The function must run `assert` on the boolean expression, resulting in program abortion (due to an `AssertionError` exception) if the test failed.

The main program can routinely run the verification test prior to solving the real problem:

```
test_solver_three_steps()
plot_numerical_and_exact(I=1, a=2, T=8, dt=0.8, theta=1)
show()
```

(Rather than calling `test_*()` functions explicitly, one will normally ask a testing framework like nose or pytest to find and run such functions.) The complete program including the verification above is found in the file `decay_v3.py`.

2.3 Computing the numerical error as a mesh function

Now that we have some evidence for a correct implementation, we are in position to compare the computed u^n values in the `u` array with the exact u values at the mesh points, in order to study the error in the numerical solution.

A natural way to compare the exact and discrete solutions is to calculate their difference as a mesh function for the error:

$$e^n = u_{\mathrm{e}}(t_n) - u^n, \quad n = 0, 1, \ldots, N_t\,. \tag{46}$$

We may view the mesh function $u^n_{\mathrm{e}} = u_{\mathrm{e}}(t_n)$ as a representation of the continuous function $u_{\mathrm{e}}(t)$ defined for all $t \in [0, T]$. In fact, u^n_{e} is often called the *representative* of u_{e} on the mesh. Then, $e^n = u^n_{\mathrm{e}} - u^n$ is clearly the difference of two mesh functions.

The error mesh function e^n can be computed by

```
u, t = solver(I, a, T, dt, theta)  # Numerical sol.
u_e = exact_solution(t, I, a)      # Representative of exact sol.
e = u_e - u
```

Note that the mesh functions `u` and `u_e` are represented by arrays and associated with the points in the array `t`.

Array arithmetics.

The last statements

```
u_e = exact_solution(t, I, a)
e = u_e - u
```

demonstrate some standard examples of array arithmetics: `t` is an array of mesh points that we pass to `exact_solution`. This function evaluates `-a*t`, which is a scalar times an array, meaning that the scalar is multiplied with each array element. The result is an array, let us call it `tmp1`. Then `exp(tmp1)` means applying the exponential function to each element in `tmp1`, giving an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` refers to this array returned from `exact_solution`. The expression `u_e - u` is the difference between two arrays, resulting in a new array referred to by `e`.

Replacement of array element computations inside a loop by array arithmetics is known as *vectorization*.

2.4 Computing the norm of the error mesh function

Instead of working with the error e^n on the entire mesh, we often want a single number expressing the size of the error. This is obtained by taking the norm of the error function.

Let us first define norms of a function $f(t)$ defined for all $t \in [0,T]$. Three common norms are

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt\right)^{1/2}, \tag{47}$$

$$||f||_{L^1} = \int_0^T |f(t)| dt, \tag{48}$$

$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,. \tag{49}$$

The L^2 norm (47) ("L-two norm") has nice mathematical properties and is the most popular norm. It is a generalization of the well-known Eucledian norm of vectors to functions. The L^1 norm looks simpler and more intuitive, but has less nice mathematical properties compared to the two other norms, so it is much less used in computations. The L^∞ is also called the max norm or the supremum norm and is widely used. It focuses on a single point with the largest value of $|f|$, while the other norms measure average behavior of the function.

In fact, there is a whole family of norms,

$$||f||_{L^p} = \left(\int_0^T f(t)^p dt\right)^{1/p}, \tag{50}$$

with p real. In particular, $p = 1$ corresponds to the L^1 norm above while $p = \infty$ is the L^∞ norm.

Numerical computations involving mesh functions need corresponding norms. Given a set of function values, f^n, and some associated mesh points, t_n, a

numerical integration rule can be used to calculate the L^2 and L^1 norms defined above. Imagining that the mesh function is extended to vary linearly between the mesh points, the Trapezoidal rule is in fact an exact integration rule. A possible modification of the L^2 norm for a mesh function f^n on a uniform mesh with spacing Δt is therefore the well-known Trapezoidal integration formula

$$||f^n|| = \left(\Delta t \left(\frac{1}{2}(f^0)^2 + \frac{1}{2}(f^{N_t})^2 + \sum_{n=1}^{N_t-1}(f^n)^2\right)\right)^{1/2}$$

A common approximation of this expression, motivated by the convenience of having a simpler formula, is

$$||f^n||_{\ell^2} = \left(\Delta t \sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}.$$

This is called the discrete L^2 norm and denoted by ℓ^2. If $||f||_{\ell^2}^2$ (i.e., the square of the norm) is used instead of the Trapezoidal integration formula, the error is $\Delta t((f^0)^2 + (f^{N_t})^2)/2$. This means that the weights at the end points of the mesh function are perturbed, but as $\Delta t \to 0$, the error from this perturbation goes to zero. As long as we are consistent and stick to one kind of integration rule for the norm of a mesh function, the details and accuracy of this rule is of no concern.

The three discrete norms for a mesh function f^n, corresponding to the L^2, L^1, and L^∞ norms of $f(t)$ defined above, are defined by

$$||f^n||_{\ell^2} = \left(\Delta t \sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}, \tag{51}$$

$$||f^n||_{\ell^1} = \Delta t \sum_{n=0}^{N_t}|f^n| \tag{52}$$

$$||f^n||_{\ell^\infty} = \max_{0\le n\le N_t}|f^n|. \tag{53}$$

Note that the L^2, L^1, ℓ^2, and ℓ^1 norms depend on the length of the interval of interest (think of $f = 1$, then the norms are proportional to $\sqrt{T}$ or T). In some applications it is convenient to think of a mesh function as just a vector of function values without any relation to the interval $[0, T]$. Then one can replace Δt by T/N_t and simply drop T (which is just a common scaling factor in the norm, independent of the vector of function values). Moreover, people prefer to divide by the total length of the vector, $N_t + 1$, instead of N_t. This reasoning gives rise to the *vector norms* for a vector $f = (f_0, \ldots, f_N)$:

$$||f||_2 = \left(\frac{1}{N+1} \sum_{n=0}^{N} (f_n)^2 \right)^{1/2}, \tag{54}$$

$$||f||_1 = \frac{1}{N+1} \sum_{n=0}^{N} |f_n| \tag{55}$$

$$||f||_{\ell^\infty} = \max_{0 \leq n \leq N} |f_n| \,. \tag{56}$$

Here we have used the common vector component notation with subscripts (f_n) and N as length. We will mostly work with mesh functions and use the discrete ℓ^2 norm (51) or the max norm ℓ^∞ (53), but the corresponding vector norms (54)-(56) are also much used in numerical computations, so it is important to know the different norms and the relations between them.

A single number that expresses the size of the numerical error will be taken as $||e^n||_{\ell^2}$ and called E:

$$E = \sqrt{\Delta t \sum_{n=0}^{N_t} (e^n)^2} \tag{57}$$

The corresponding Python code, using array arithmetics, reads

```
E = sqrt(dt*sum(e**2))
```

The `sum` function comes from `numpy` and computes the sum of the elements of an array. Also the `sqrt` function is from `numpy` and computes the square root of each element in the array argument.

Scalar computing. Instead of doing array computing `sqrt(dt*sum(e**2))` we can compute with one element at a time:

```
m = len(u)     # length of u array (alt: u.size)
u_e = zeros(m)
t = 0
for i in range(m):
    u_e[i] = exact_solution(t, a, I)
    t = t + dt
e = zeros(m)
for i in range(m):
    e[i] = u_e[i] - u[i]
s = 0  # summation variable
for i in range(m):
    s = s + e[i]**2
error = sqrt(dt*s)
```

Such element-wise computing, often called *scalar* computing, takes more code, is less readable, and runs much slower than what we can achieve with array computing.

2.5 Experiments with computing and plotting

Let us write down a new function that wraps up the computation and all the plotting statements used for comparing the exact and numerical solutions. This function can be called with various θ and Δt values to see how the error depends on the method and mesh resolution.

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).
    """
    u, t = solver(I, a, T, dt, theta)    # Numerical solution
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                         # create new plot
        t_e = linspace(0, T, 1001)       # fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'r--o')           # red dashes w/circles
        plot(t_e, u_e, 'b-')             # blue line for exact sol.
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        show()
    return E
```

The `figure()` call is key: without it, a new `plot` command will draw the new pair of curves in the same plot window, while we want the different pairs to appear in separate windows and files. Calling `figure()` ensures this.

Instead of including the θ value in the filename to implicitly inform about the applied method, the code utilizes a little Python dictionary that maps each relevant θ value to a corresponding acronym for the method name (FE, BE, or CN):

```
theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
savefig('%s_%g.png' % (theta2name[theta], dt))
```

The `explore` function stores the plot in two different image file formats: PNG and PDF. The PNG format is suitable for being included in HTML documents, while the PDF format provides higher quality for LaTeX (i.e., pdfLaTeX) documents. Frequently used viewers for these image files on Unix systems are `gv` (comes with Ghostscript) for the PDF format and `display` (from the ImageMagick software suite) for PNG files:

```
Terminal> gv BE_0.5.pdf
Terminal> display BE_0.5.png
```

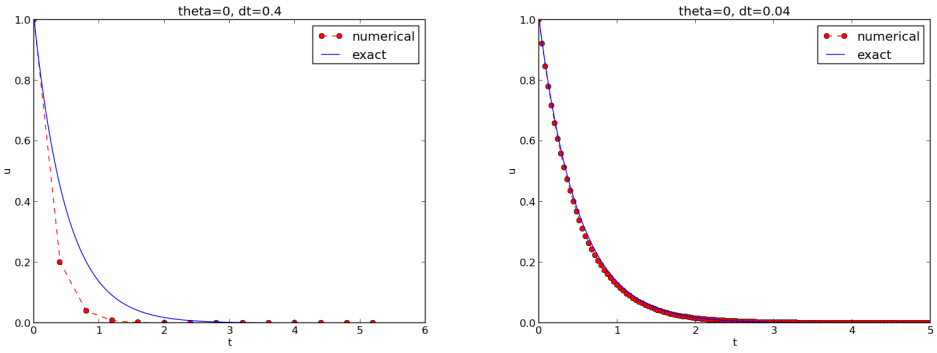


Figure 7: The Forward Euler scheme for two values of the time step.

A main program may run a loop over the three methods (given by their corresponding θ values) and call `explore` to compute errors and make plots:

```
def main(I, a, T, dt_values, theta_values=(0, 0.5, 1)):
    print 'theta   dt       error'  # Column headings in table
    for theta in theta_values:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot=True)
            print '%4.1f %6.2f: %12.3E' % (theta, dt, E)

main(I=1, a=2, T=5, dt_values=[0.4, 0.04])
```

The file `decay_plot_mpl.py` contains the complete code with the functions above. Running this program results in

```
Terminal> python decay_plot_mpl.py
theta   dt       error
 0.0   0.40:    2.105E-01
 0.0   0.04:    1.449E-02
 0.5   0.40:    3.362E-02
 0.5   0.04:    1.887E-04
 1.0   0.40:    1.030E-01
 1.0   0.04:    1.382E-02
```

We observe that reducing Δt by a factor of 10 increases the accuracy for all three methods. We also see that the combination of $\theta = 0.5$ and a small time step $\Delta t = 0.04$ gives a much more accurate solution, and that $\theta = 0$ and $\theta = 1$ with $\Delta t = 0.4$ result in the least accurate solutions.

Figure 7 demonstrates that the numerical solution produced by the Forward Euler method with $\Delta t = 0.4$ clearly lies below the exact curve, but that the accuracy improves considerably by reducing the time step by a factor of 10.

The behavior of the two other schemes is shown in Figures 8 and 9. Crank-Nicolson is obviously the most accurate scheme from this visual point of view.

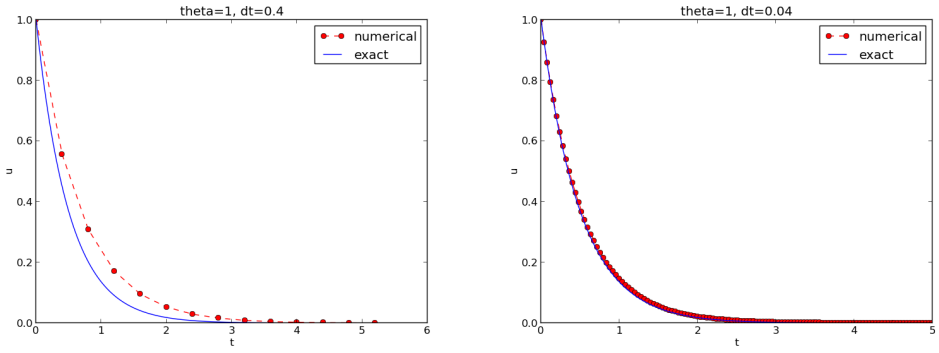


Figure 8: The Backward Euler scheme for two values of the time step.

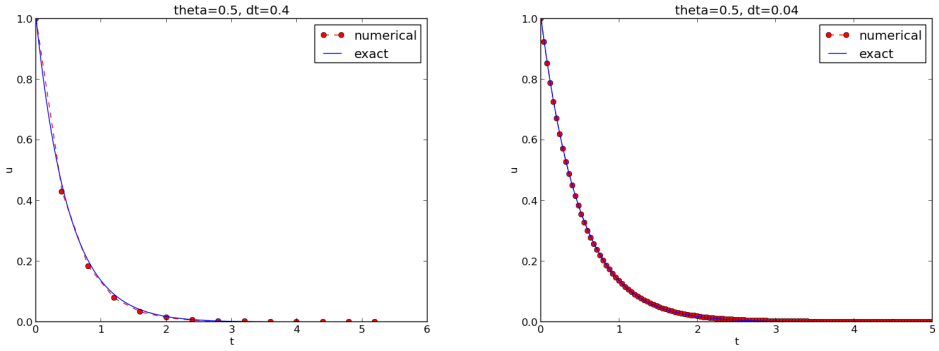


Figure 9: The Crank-Nicolson scheme for two values of the time step.

Combining plot files. Mounting two PNG files beside each other, as done in Figures 7-9, is easily carried out by the `montage` program from the ImageMagick suite:

```
Terminal> montage -background white -geometry 100% -tile 2x1 \
          FE_0.4.png FE_0.04.png FE1.png
Terminal> convert -trim FE1.png FE1.png
```

The `-geometry` argument is used to specify the size of the image. Here, we preserve the individual sizes of the images. The `-tile HxV` option specifies `H` images in the horizontal direction and `V` images in the vertical direction. A series of image files to be combined are then listed, with the name of the resulting combined image, here `FE1.png` at the end. The `convert -trim` command removes surrounding white areas in the figure (an operation usually known as *cropping* in image manipulation programs).

For LaTeX reports it is not recommended to use `montage` and PNG files as the result has too low resolution. Instead, plots should be made in the PDF format and combined using the `pdftk`, `pdfnup`, and `pdfcrop` tools (on Linux/Unix):

```
Terminal> pdftk FE_0.4.png FE_0.04.png output tmp.pdf
Terminal> pdfnup --nup 2x1 --outfile tmp.pdf tmp.pdf
Terminal> pdfcrop tmp.pdf FE1.png  # output in FE1.png
```

Here, `pdftk` combines images into a multi-page PDF file, `pdfnup` combines the images in individual pages to a table of images (pages), and `pdfcrop` removes white margins in the resulting combined image file.

Plotting with SciTools. The SciTools package provides a unified plotting interface, called Easyviz, to many different plotting packages, including Matplotlib, Gnuplot, Grace, MATLAB, VTK, OpenDX, and VisIt. The syntax is very similar to that of Matplotlib and MATLAB. In fact, the plotting commands shown above look the same in SciTool's Easyviz interface, apart from the import statement, which reads

```
from scitools.std import *
```

This statement performs a `from numpy import *` as well as an import of the most common pieces of the Easyviz (`scitools.easyviz`) package, along with some additional numerical functionality.

With Easyviz one can merge several plotting commands into a single one using keyword arguments:

```
plot(t,   u,   'r--o',           # red dashes w/circles
     t_e, u_e, 'b-',             # blue line for exact sol.
     legend=['numerical', 'exact'],
     xlabel='t',
     ylabel='u',
     title='theta=%g, dt=%g' % (theta, dt),
     savefig='%s_%g.png' % (theta2name[theta], dt),
     show=True)
```

The `decay_plot_st.py` file contains such a demo.

By default, Easyviz employs Matplotlib for plotting, but Gnuplot and Grace are viable alternatives:

```
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend gnuplot
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend grace
```

The actual tool used for creating plots (called *backend*) and numerous other options can be permanently set in SciTool's configuration file.

All the Gnuplot windows are launched without any need to kill one before the next one pops up (as is the case with Matplotlib) and one can press the key 'q' anywhere in a plot window to kill it. Another advantage of Gnuplot is the automatic choice of sensible and distinguishable line types in black-and-white PDF and PostScript files.

For more detailed information on syntax and plotting capabilities, we refer to the Matplotlib [4] and SciTools [6] documentation. The hope is that the programming syntax explained so far suffices for understanding the basic plotting functionality and being able to look up the cited technical documentation.

Test your understanding.

Exercise 18 asks you to implement a solver for a problem that is slightly different from the one above. You may use the `solver` and `explore` functions explained above as a starting point. Apply the new solver to solve Exercise 19.

2.6 Memory-saving implementation

The computer memory requirements of our implementations so far consist mainly of the `u` and `t` arrays, both of length $N_t + 1$. Also, for the programs that involve array arithmetics, Python needs memory space for storing temporary arrays. For example, computing `I*exp(-a*t)` requires storing the intermediate result `a*t` before the preceding minus sign can be applied. The resulting array is temporarily stored and provided as input to the `exp` function. Regardless of how we implement simple ODE problems, storage requirements are very modest and put no restrictions on how we choose our data structures and algorithms. Nevertheless, when the presented methods are applied to three-dimensional PDE problems, memory storage requirements suddenly become a challenging issue.

Let us briefly elaborate on how large the storage requirements can quickly be in three-dimensional problems. The PDE counterpart to our model problem $u' = -a$ is a diffusion equation $u_t = a\nabla^2 u$ posed on a space-time domain. The discrete representation of this domain may in 3D be a spatial mesh of M^3 points and a time mesh of N_t points. In many applications, it is quite typical that M is at least 100, or even 1000. Storing all the computed u values, like we have done in the programs so far, would demand storing arrays of size up to $M^3 N_t$. This would give a factor of M^3 larger storage demands compared to what was required by our ODE programs. Each real number in the `u` array requires 8 bytes (b) of storage. With $M = 100$ and $N_t = 1000$, there is a storage demand of $(10^3)^3 \cdot 1000 \cdot 8 = 8$ Gb for the solution array. Fortunately, we can usually get rid of the N_t factor, resulting in 8 Mb of storage. Below we explain how this is done (the technique is almost always applied in implementations of PDE problems).

Let us critically evaluate how much we really need to store in the computer's memory for our implementation of the θ method. To compute a new u^{n+1}, all we need is u^n. This implies that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do not need to be stored, although this is convenient for plotting and data analysis in the program. Instead of the `u` array we can work with two variables for real numbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorithm, respectively. At each time level, we update `u` from `u_1` and then set `u_1 = u`, so that the computed u^{n+1} value becomes the "previous" value u^n at the next time level. The downside is that we cannot plot the solution after the simulation is done since only the last two numbers are available. The remedy is to store computed values in a file and use the file for visualizing the solution later.

We have implemented this memory saving idea in the file `decay_memsave.py`, which is a slight modification of `decay_plot_mpl.py` program.

The following function demonstrates how we work with the two most recent values of the unknown:

```
def solver_memsave(I, a, T, dt, theta, filename='sol.dat'):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.
    Minimum use of memory. The solution is stored in a file
    (with name filename) for later plotting.
    """
    dt = float(dt)         # avoid integer division
    Nt = int(round(T/dt))  # no of intervals

    outfile = open(filename, 'w')
    # u: time level n+1, u_1: time level n
    t = 0
    u_1 = I
    outfile.write('%.16E  %.16E\n' % (t, u_1))
    for n in range(1, Nt+1):
        u = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u_1
        u_1 = u
        t += dt
        outfile.write('%.16E  %.16E\n' % (t, u))
    outfile.close()
    return u, t
```

This code snippet also serves as a quick introduction to file writing in Python. Reading the data in the file into arrays `t` and `u` are done by the function

```
def read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = [];  t = []
    for line in infile:
        words = line.split()
        if len(words) != 2:
            print 'Found more than two numbers on a line!', words
            sys.exit(1)  # abort
        t.append(float(words[0]))
        u.append(float(words[1]))
    return np.array(t), np.array(u)
```

This type of file with numbers in rows and columns is very common, and `numpy` has a function `loadtxt` which loads such tabular data into a two-dimensional

array named by the user. Say the name is `data`, the number in row `i` and column `j` is then `data[i,j]`. The whole column number `j` can be extracted by `data[:,j]`. A version of `read_file` using `np.loadtxt` reads

```
def read_file_numpy(filename='sol.dat'):
    data = np.loadtxt(filename)
    t = data[:,0]
    u = data[:,1]
    return t, u
```

The present counterpart to the `explore` function from `decay_plot_mpl.py` must run `solver_memsave` and then load data from file before we can compute the error measure and make the plot:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    filename = 'u.dat'
    u, t = solver_memsave(I, a, T, dt, theta, filename)

    t, u = read_file(filename)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*np.sum(e**2))
    if makeplot:
        figure()
        ...
```

Apart from the internal implementation, where u^n values are stored in a file rather than in an array, `decay_memsave.py` file works exactly as the `decay_plot_mpl.py` file.

3 Exercises

Exercise 1: Define a mesh function and visualize it

a) Write a function `mesh_function(f, t)` that returns an array with mesh point values $f(t_0), \ldots, f(t_{N_t})$, where `f` is a Python function implementing a mathematical function `f(t)` and $t_0, \ldots, t_{N_t}$ are mesh points stored in the array `t`. Use a loop over the mesh points and compute one mesh function value at the time.

b) Use `mesh_function` to compute the mesh function corresponding to

$$f(t) = \begin{cases} e^{-t}, & 0 \leq t \leq 3, \\ e^{-3t}, & 3 < t \leq 4 \end{cases}$$

Choose a mesh $t_n = n\Delta t$ with $\Delta t = 0.1$. Plot the mesh function.
Filename: `mesh_function`.

Remarks. In Section 2.3 we show how easy it is to compute a mesh function by array arithmetics (or array computing). Using this technique, one could simply implement `mesh_function(f,t)` as `return f(t)`. However, `f(t)` will not work if there are if tests involving `t` inside `f` as is the case in b). Typically, `if t < 3` must have `t < 3` as a boolean expression, but if `t` is array, `t < 3`, is an *array of boolean values*, which is not legal as a boolean expression in an if test. Computing one element at a time as suggested in a) is a way of out of this problem.

We also remark that the function in b) is the solution of $u' = -au$, $u(0) = 1$, for $t \in [0, 4]$, where $a = 1$ for $t \in [0, 3]$ and $a = 3$ for $t \in [3, 4]$.

Exercise 2: Differentiate a function

Given a mesh function u^n as an array `u` with u^n values at mesh points $t_n = n\Delta t$, the discrete derivative can be based on centered differences:

$$d^n = [D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}, \quad n = 1, \ldots, N_t - 1\,. \tag{58}$$

At the end points we use forward and backward differences:

$$d^0 = [D_t^+ u]^n = \frac{u^1 - u^0}{\Delta t},$$

and

$$d^{N_t} = [D_t^- u]^n = \frac{u^{N_t} - u^{N_t-1}}{\Delta t}\,.$$

a) Write a function `differentiate(u, dt)` that returns the discrete derivative d^n of the mesh function u^n. The parameter `dt` reflects the mesh spacing Δt. Write a corresponding test function `test_differentiate()` for verifying the implementation.

Hint. The three differentiation formulas are exact for quadratic polynomials. Use this property to verify the program.

b) A standard implementation of the formula (58) is to have a loop over i. For large N_t, such loop may run slowly in Python. A technique for speeding up the computations, called vectorization or array computing, replaces the loop by array operations. To see how this can be done in the present mathematical problem, we define two arrays

$$u^+ = (u^2, u^3, \ldots, u^{N_t}), u^- \qquad = (u^0, u^1, \ldots, u^{N_t-2})\,.$$

The formula (58) can now be expressed as

$$(d^1, d^2, \ldots, d^{N_t-1}) = \frac{1}{2\Delta t}(u^+ - u^-)\,.$$

The corresponding Python code reads

```
d[1:-1] = (u[2:] - u[0:-2])/(2*dt)
# or
d[1:N_t] = (u[2:N_t+1] - u[0:N_t-1])/(2*dt)
```

Recall that an array slice `u[1:-1]` contains the elements in `u` starting with index 1 and going all indices up to, but not including, the last one (`-1`).

Use the ideas above to implement a vectorized version of the `differentiate` function without loops.
Filename: `differentiate`.

Exercise 3: Experiment with integer division

Explain what happens in the following computations, where some are mathematically unexpected:

```
>>> dt = 3
>>> T = 8
>>> Nt = T/dt
>>> Nt
2
>>> theta = 1; a = 1
>>> (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
0
```

Filename: `pyproblems.txt`.

Exercise 4: Experiment with wrong computations

Consider the `solver` function in the `decay_v1.py` file and the following call:

```
u, t = solver(I=1, a=1, T=7, dt=2, theta=1)
```

The output becomes

```
t= 0.000 u=1
t= 2.000 u=0
t= 4.000 u=0
t= 6.000 u=0
```

Print out the result of all intermediate computations and use `type(v)` to see the object type of the result stored in `v`. Examine the intermediate calculations and explain why `u` is wrong and why we compute up to $t = 6$ only even though we specified $T = 7$. Filename: `decay_v1_err`.

Exercise 5: Plot the error function

Solve the problem $u' = -au$, $u(0) = I$, using the Forward Euler, Backward Euler, and Crank-Nicolson schemes. For each scheme, plot the error mesh function $e^n = u_{\mathrm{e}}(t_n) - u^n$ for Δt, $\frac{1}{4}\Delta t$, and $\frac{1}{8}\Delta t$, where u_{e} is the exact solution of the ODE and u^n is the numerical solution at mesh point t_n. Filename: `decay_plot_error`.

Exercise 6: Change formatting of numbers and debug

The `decay_memsave.py` program writes the time values and solution values to a file which looks like

```
0.0000000000000000E+00  1.0000000000000000E+00
2.0000000000000001E-01  8.3333333333333337E-01
4.0000000000000002E-01  6.9444444444444453E-01
6.0000000000000009E-01  5.7870370370370383E-01
8.0000000000000004E-01  4.8225308641975323E-01
1.0000000000000000E+00  4.0187757201646102E-01
1.2000000000000000E+00  3.3489797668038418E-01
1.3999999999999999E+00  2.7908164723365347E-01
```

Modify the file output such that it looks like

```
0.000  1.00000
0.200  0.83333
0.400  0.69444
0.600  0.57870
0.800  0.48225
1.000  0.40188
1.200  0.33490
1.400  0.27908
```

If you have just modified the formatting of numbers in the file, running the modified program

```
Terminal> python decay_memsave_v2.py --T 10 --theta 1 \
          --dt 0.2 --makeplot
```

leads to printing of the message `Bug in the implementation!` in the terminal window. Why?

Answer. With only 5 decimals in the file, the `test_solver_minmem` function compares truncated elements `u`, accurate only to 10^{-5} with the exact discrete solution and applies a far too small `tol` value. `tol` must be `1E-4`.
Filename: `decay_memsave_v2`.

4 Analysis of finite difference equations

We address the ODE for exponential decay,

$$u'(t) = -au(t), \quad u(0) = I, \tag{59}$$

where a and I are given constants. This problem is solved by the θ-rule finite difference scheme, resulting in the recursive equations

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n \tag{60}$$

for the numerical solution u^{n+1}, which approximates the exact solution u_{e} at time point t_{n+1}. For constant mesh spacing, which we assume here, $t_{n+1} = (n+1)\Delta t$.

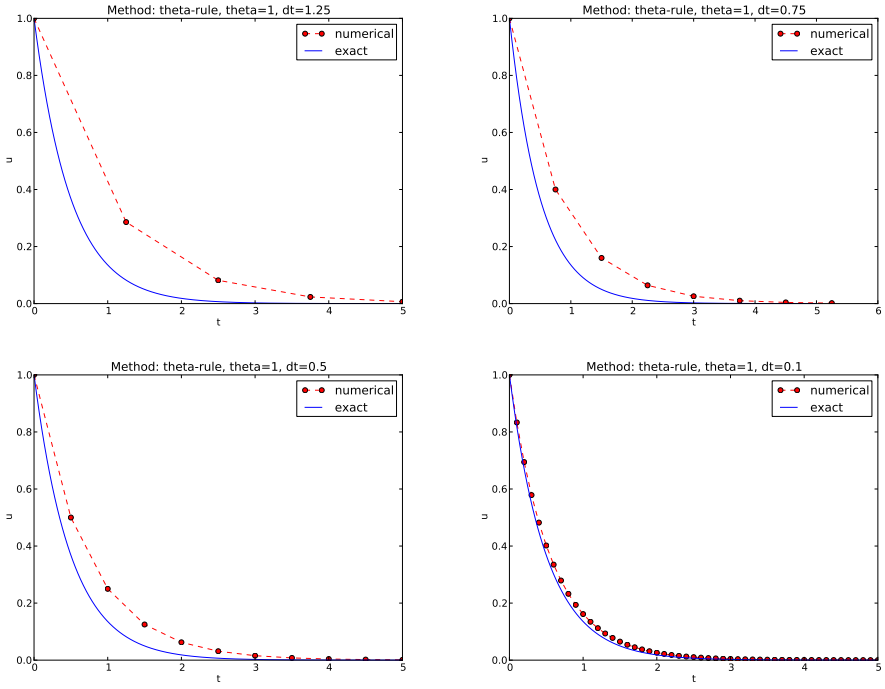


Figure 10: Backward Euler.

Discouraging numerical solutions. Choosing $I = 1$, $a = 2$, and running experiments with $\theta = 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the results in Figures 10, 11, and 12.

The characteristics of the displayed curves can be summarized as follows:

- The Backward Euler scheme gives a monotone solution in all cases, lying above the exact curve.
- The Crank-Nicolson scheme gives the most accurate results, but for $\Delta t = 1.25$ the solution oscillates.
- The Forward Euler scheme gives a growing, oscillating solution for $\Delta t = 1.25$; a decaying, oscillating solution for $\Delta t = 0.75$; a strange solution $u^n = 0$ for $n \geq 1$ when $\Delta t = 0.5$; and a solution seemingly as accurate as the one by the Backward Euler scheme for $\Delta t = 0.1$, but the curve lies below the exact solution.

Since the exact solution of our model problem is a monotone function, $u(t) = Ie^{-at}$, some of these qualitatively wrong results indeed seem alarming!

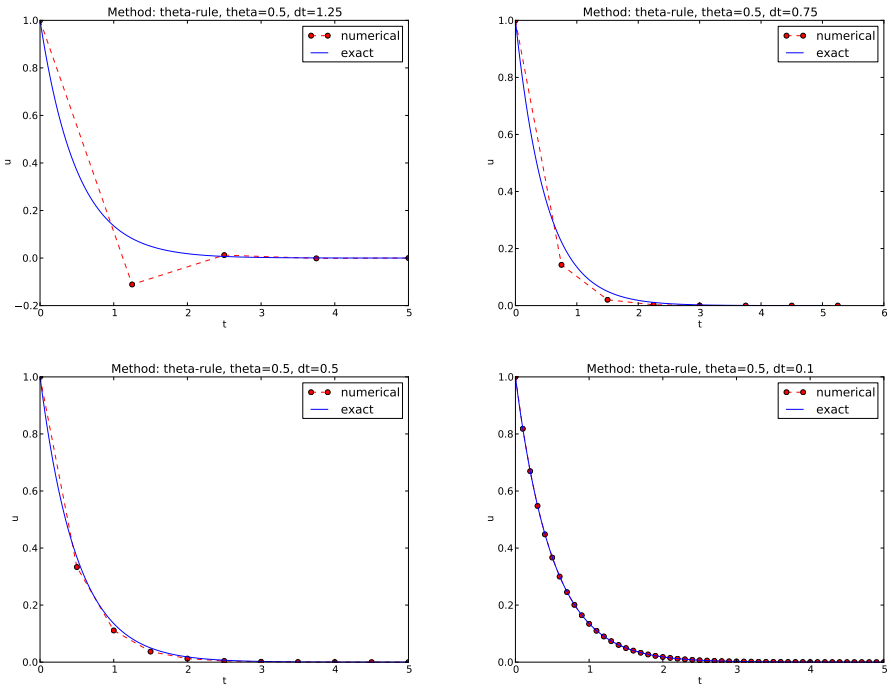


Figure 11: Crank-Nicolson.

Key questions.

- Under what circumstances, i.e., values of the input data I, a, and Δt will the Forward Euler and Crank-Nicolson schemes result in undesired oscillatory solutions?

- How does Δt impact the error in the numerical solution?

The first question will be investigated both by numerical experiments and by precise mathematical theory. The theory will help establish general criteria on Δt for avoiding non-physical oscillatory or growing solutions.

For our simple model problem we can answer the second question very precisely, but we will also look at simplified formulas for small Δt and touch upon important concepts such as *convergence rate* and *the order of a scheme*. Other fundamental concepts mentioned are stability, consistency, and convergence.

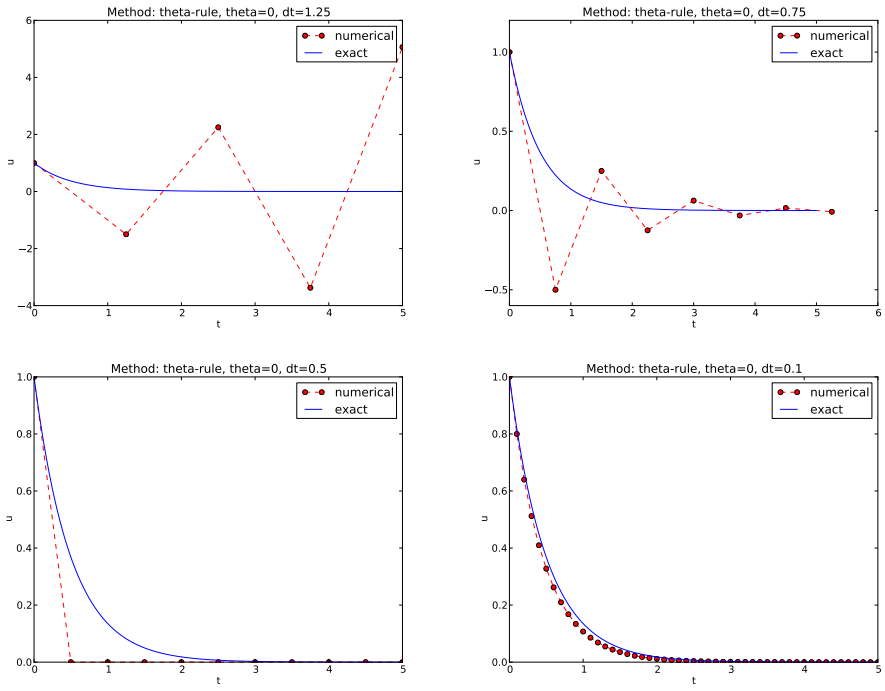


Figure 12: Forward Euler.

4.1 Experimental investigation of oscillatory solutions

To address the first question above, we may set up an experiment where we loop over values of I, a, and Δt in our chosen model problem. For each experiment, we flag the solution as oscillatory if

$$u^n > u^{n-1},$$

for some value of n. This seems like a reasonable choice, since we expect u^n to decay with n, but oscillations will make u increase over a time step. Doing some initial experimentation with varying I, a, and Δt, quickly reveals that oscillations are independent of I, but they do depend on a and Δt. We can therefore limit the investigation to vary a and Δt. Based on this observation, we introduce a two-dimensional function $B(a, \Delta t)$ which is 1 if oscillations occur and 0 otherwise. We can visualize B as a contour plot (lines for which $B = \text{const}$). The contour $B = 0.5$ corresponds to the borderline between oscillatory regions with $B = 1$ and monotone regions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have given P values for a, $a_0, \ldots, a_{P-1}$, and Q values for Δt, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, $i = 0, \ldots, P-1$, $j = 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the plane spanned by a and Δt. At each point $(a_i, \Delta t_j)$, we

associate the corresponding value $B(a_i, \Delta t_j)$, denoted B_{ij}. The B_{ij} values are naturally stored in a two-dimensional array. We can thereafter create a plot of the contour line $B_{ij} = 0.5$ dividing the oscillatory and monotone regions. The file `decay_osc_regions.py` given below (`osc_regions` stands for "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ line in Figures 13 and 14. The oscillatory region is above this line.

```
from decay_mod import solver
import numpy as np
import scitools.std as st

def non_physical_behavior(I, a, T, dt, theta):
    """
    Given lists/arrays a and dt, and numbers I, dt, and theta,
    make a two-dimensional contour line B=0.5, where B=1>0.5
    means oscillatory (unstable) solution, and B=0<0.5 means
    monotone solution of u'=-au.
    """
    a = np.asarray(a); dt = np.asarray(dt)  # must be arrays
    B = np.zeros((len(a), len(dt)))         # results
    for i in range(len(a)):
        for j in range(len(dt)):
            u, t = solver(I, a[i], T, dt[j], theta)
            # Does u have the right monotone decay properties?
            correct_qualitative_behavior = True
            for n in range(1, len(u)):
                if u[n] > u[n-1]:  # Not decaying?
                    correct_qualitative_behavior = False
                    break  # Jump out of loop
            B[i,j] = float(correct_qualitative_behavior)
    a_, dt_ = st.ndgrid(a, dt)  # make mesh of a and dt values
    st.contour(a_, dt_, B, 1)
    st.grid('on')
    st.title('theta=%g' % theta)
    st.xlabel('a'); st.ylabel('dt')
    st.savefig('osc_region_theta_%s.png' % theta)
    st.savefig('osc_region_theta_%s.pdf' % theta)

non_physical_behavior(
    I=1,
    a=np.linspace(0.01, 4, 22),
    dt=np.linspace(0.01, 4, 22),
    T=6,
    theta=0.5)
```

By looking at the curves in the figures one may guess that $a\Delta t$ must be less than a critical limit to avoid the undesired oscillations. This limit seems to be about 2 for Crank-Nicolson and 1 for Forward Euler. We shall now establish a precise mathematical analysis of the discrete model that can explain the observations in our numerical experiments.

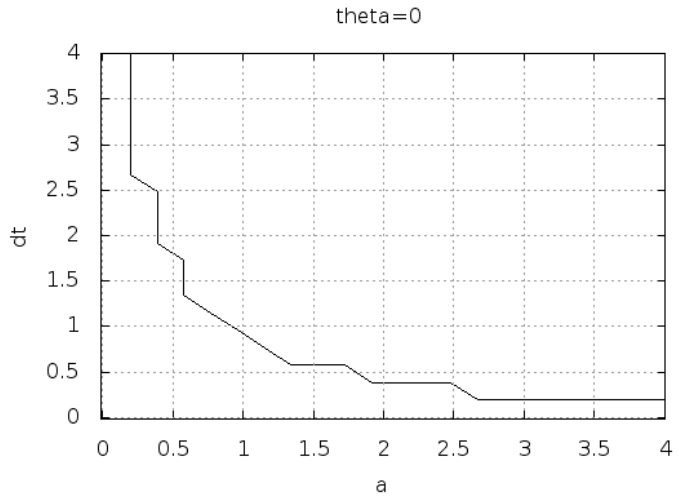


Figure 13: Forward Euler scheme: oscillatory solutions occur for points above the curve.

4.2 Exact numerical solution

Starting with $u^0 = I$, the simple recursion (60) can be applied repeatedly n times, with the result that

$$u^n = IA^n, \quad A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,. \tag{61}$$

Solving difference equations.

Difference equations where all terms are linear in u^{n+1}, u^n, and maybe u^{n-1}, u^{n-2}, etc., are called *homogeneous, linear* difference equations, and their solutions are generally of the form $u^n = A^n$, where A is a constant to be determined. Inserting this expression in the difference equation and dividing by A^{n+1} gives a polynomial equation in A. In the present case we get

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,.$$

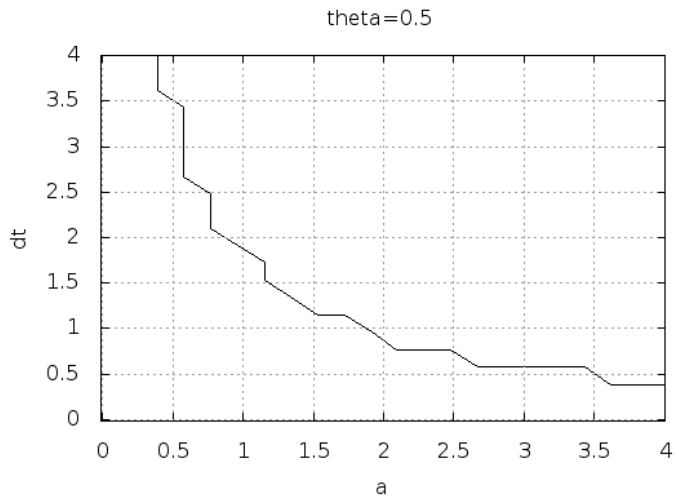


Figure 14: Crank-Nicolson scheme: oscillatory solutions occur for points above the curve.

This is a solution technique of wider applicability than repeated use of the recursion (60).

Regardless of the solution approach, we have obtained a formula for u^n. This formula can explain everything we see in the figures above, but it also gives us a more general insight into accuracy and stability properties of the three schemes.

4.3 Stability

Since u^n is a factor A raised to an integer power n, we realize that $A < 0$ will imply $u^n < 0$ for odd n and $u^n > 0$ for even n. That is, the solution oscillates between the mesh points. We have oscillations due to $A < 0$ when

$$(1-\theta)a\Delta t > 1\,. \tag{62}$$

Since $A > 0$ is a requirement for having a numerical solution with the same basic property (monotonicity) as the exact solution, we may say that $A > 0$ is a *stability criterion*. Expressed in terms of Δt the stability criterion reads

$$\Delta t < \frac{1}{(1-\theta)a}\,. \tag{63}$$

The Backward Euler scheme is always stable since $A < 0$ is impossible for $\theta = 1$, while non-oscillating solutions for Forward Euler and Crank-Nicolson demand $\Delta t \leq 1/a$ and $\Delta t \leq 2/a$, respectively. The relation between Δt and a look reasonable: a larger a means faster decay and hence a need for smaller time steps.

Looking at the upper left plot in Figure 12, we see that $\Delta t = 1.25$, and remembering that $a = 2$ in these experiments, A can be calculated to be -1.5, so the Forward Euler solution becomes $u^n = (-1.5)^n$ ($I = 1$). This solution oscillates *and* grows. The upper right plot has $a\Delta t = 2 \cdot 0.75 = 1.5$, so $A = -0.5$, and $u^n = (-0.5)^n$ decays but oscillates. The lower left plot is a peculiar case where the Forward Euler scheme produces a solution that is stuck on the t axis. Now we can understand why this is so, because $a\Delta t = 2 \cdot 0.5 = 1$, which gives $A = 0$, and therefore $u^n = 0$ for $n \geq 1$. The decaying oscillations in the Crank-Nicolson scheme in the upper left plot in Figure 11 for $\Delta t = 1.25$ are easily explained by the fact that $A \approx -0.11 < 0$.

The factor A is called the *amplification factor* since the solution at a new time level is A times the solution at the previous time level. For a decay process, we must obviously have $|A| \leq 1$, which is fulfilled for all Δt if $\theta \geq 1/2$. Arbitrarily large values of u can be generated when $|A| > 1$ and n is large enough. The numerical solution is in such cases totally irrelevant to an ODE modeling decay processes! To avoid this situation, we must for $\theta < 1/2$ have

$$\Delta t \leq \frac{2}{(1-2\theta)a}, \tag{64}$$

which means $\Delta t < 2/a$ for the Forward Euler scheme.

Stability properties.

We may summarize the stability investigations as follows:

1. The Forward Euler method is a *conditionally stable* scheme because it requires $\Delta t < 2/a$ for avoiding growing solutions and $\Delta t < 1/a$ for avoiding oscillatory solutions.
2. The Crank-Nicolson is *unconditionally stable* with respect to growing solutions, while it is conditionally stable with the criterion $\Delta t < 2/a$ for avoiding oscillatory solutions.
3. The Backward Euler method is unconditionally stable with respect to growing and oscillatory solutions - any Δt will work.

Much literature on ODEs speaks about L-stable and A-stable methods. In our case A-stable methods ensures non-growing solutions, while L-stable methods also avoids oscillatory solutions.

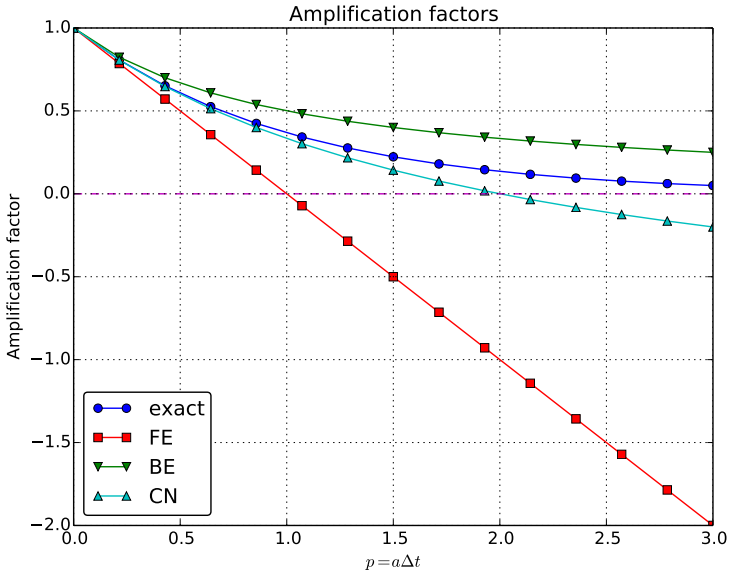


Figure 15: Comparison of amplification factors.

4.4 Comparing amplification factors

After establishing how A impacts the qualitative features of the solution, we shall now look more into how well the numerical amplification factor approximates the exact one. The exact solution reads $u(t) = Ie^{-at}$, which can be rewritten as

$$u_{\mathrm{e}}(t_n) = Ie^{-an\Delta t} = I(e^{-a\Delta t})^n\,. \tag{65}$$

From this formula we see that the exact amplification factor is

$$A_{\mathrm{e}} = e^{-a\Delta t}\,. \tag{66}$$

We see from all of our analysis that the exact and numerical amplification factors depend on a and Δt through the dimensionless product $a\Delta t$: whenever there is a Δt in the analysis, there is always an associated a parameter. Therefore, it is convenient to introduce a symbol for this product, $p = a\Delta t$, and view A and A_{e} as functions of p. Figure 15 shows these functions. The two amplification factors are clearly closest for the Crank-Nicolson method, but that method has the unfortunate oscillatory behavior when $p > 2$.

Significance of the $p = a\Delta t$ parameter.

The key parameter for numerical performance of a scheme is in this model problem $p = a\Delta t$. This is a *dimensionless number* (a has dimension 1/s

and Δt has dimension s) reflecting how the discretization parameter plays together with a physical parameter in the problem.

One can bring the present model problem on dimensionless form (through a process called scaling, see Section 9.1 for a brief introduction and Chapter **??** in [5] for a comprehensive treatment). The scaled modeled has a modified time $\bar{t} = at$ and modified response $\bar{u} = u/I$ such that the model reads $d\bar{u}/d\bar{t} = -\bar{u}$, $\bar{u}(0) = 1$. Analyzing this model, where there are no physical parameters, we find that $\Delta\bar{t}$ is the key parameter for numerical performance. In the unscaled model, this corresponds to $\Delta\bar{t} = a\Delta t$.

It is common that the numerical performance of methods for solving ordinary and partial differential equations is governed by dimensionless parameters that combine mesh sizes with physical parameters.

4.5 Series expansion of amplification factors

As an alternative to the visual understanding inherent in Figure 15, there is a strong tradition in numerical analysis to establish formulas for approximation errors when the discretization parameter, here Δt, becomes small. In the present case, we let p be our small discretization parameter, and it makes sense to simplify the expressions for A and A_{e} by using Taylor polynomials around $p = 0$. The Taylor polynomials are accurate for small p and greatly simplifies the comparison of the analytical expressions since we then can compare polynomials, term by term.

Calculating the Taylor series for A_{e} is easily done by hand, but the three versions of A for $\theta = 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. Nowadays, analytical computations can benefit greatly by symbolic computer algebra software. The Python package `sympy` represents a powerful computer algebra system, not yet as sophisticated as the famous Maple and Mathematica systems, but it is free and very easy to integrate with our numerical computations in Python.

When using `sympy`, it is convenient to enter an interactive Python shell where the results of expressions and statements can be shown immediately. Here is a simple example. We strongly recommend to use `isympy` (or `ipython`) for such interactive sessions.

Let us illustrate `sympy` with a standard Python shell syntax (`>>>` prompt) to compute a Taylor polynomial approximation to e^{-p}:

```
>>> from sympy import *
>>> # Create p as a mathematical symbol with name 'p'
>>> p = Symbol('p')
>>> # Create a mathematical expression with p
>>> A_e = exp(-p)
>>>
>>> # Find the first 6 terms of the Taylor series of A_e
>>> A_e.series(p, 0, 6)
1 + (1/2)*p**2 - p - 1/6*p**3 - 1/120*p**5 + (1/24)*p**4 + O(p**6)
```

Lines with >>> represent input lines, whereas without this prompt represent the result of the previous command (note that `isympy` and `ipython` apply other prompts, but in this text we always apply >>> for interactive Python computing). Apart from the order of the powers, the computed formula is easily recognized as the beginning of the Taylor series for e^{-p}.

Let us define the numerical amplification factor where p and θ enter the formula as symbols:

```
>>> theta = Symbol('theta')
>>> A = (1-(1-theta)*p)/(1+theta*p)
```

To work with the factor for the Backward Euler scheme we can substitute the value 1 for `theta`:

```
>>> A.subs(theta, 1)
1/(1 + p)
```

Similarly, we can replace `theta` by 1/2 for Crank-Nicolson, preferably using an exact rational representation of 1/2 in `sympy`:

```
>>> half = Rational(1,2)
>>> A.subs(theta, half)
1/(1 + (1/2)*p)*(1 - 1/2*p)
```

The Taylor series of the amplification factor for the Crank-Nicolson scheme can be computed as

```
>>> A.subs(theta, half).series(p, 0, 4)
1 + (1/2)*p**2 - p - 1/4*p**3 + O(p**4)
```

We are now in a position to compare Taylor series:

```
>>> FE = A_e.series(p, 0, 4) - A.subs(theta, 0).series(p, 0, 4)
>>> BE = A_e.series(p, 0, 4) - A.subs(theta, 1).series(p, 0, 4)
>>> CN = A_e.series(p, 0, 4) - A.subs(theta, half).series(p, 0, 4 )
>>> FE
(1/2)*p**2 - 1/6*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (5/6)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

From these expressions we see that the error $A - A_{\mathrm{e}} \sim \mathcal{O}(p^2)$ for the Forward and Backward Euler schemes, while $A - A_{\mathrm{e}} \sim \mathcal{O}(p^3)$ for the Crank-Nicolson scheme. The notation $\mathcal{O}(p^m)$ here means a polynomial in p where p^m is the term of lowest-degree, and consequently the term that dominates the expression for $p < 0$. We call this the *leading order term*. As $p \to 0$, the leading order term clearly dominates over the higher-order terms (think of $p = 0.01$: p is a hundred times larger than p^2).

Now, a is a given parameter in the problem, while Δt is what we can vary. Not surprisingly, the error expressions are usually written in terms Δt. When then have

$$A - A_{\mathrm{e}} = \begin{cases} \mathcal{O}(\Delta t^2), & \text{Forward and Backward Euler,} \\ \mathcal{O}(\Delta t^3), & \text{Crank-Nicolson} \end{cases} \tag{67}$$

We say that the Crank-Nicolson scheme has an error in the amplification factor of order Δt^3, while the two other schemes are of order Δt^2 in the same quantity.

What is the significance of the order expression? If we halve Δt, the error in amplification factor at a time level will be reduced by a factor of 4 in the Forward and Backward Euler schemes, and by a factor of 8 in the Crank-Nicolson scheme. That is, as we reduce Δt to obtain more accurate results, the Crank-Nicolson scheme reduces the error more efficiently than the other schemes.

4.6 The fraction of numerical and exact amplification factors

An alternative comparison of the schemes is provided by looking at the ratio A/A_{e}, or the error $1 - A/A_{\mathrm{e}}$ in this ratio:

```
>>> FE = 1 - (A.subs(theta, 0)/A_e).series(p, 0, 4)
>>> BE = 1 - (A.subs(theta, 1)/A_e).series(p, 0, 4)
>>> CN = 1 - (A.subs(theta, half)/A_e).series(p, 0, 4)
>>> FE
(1/2)*p**2 + (1/3)*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (1/3)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

The leading-order terms have the same powers as in the analysis of $A - A_{\mathrm{e}}$.

4.7 The global error at a point

The error in the amplification factor reflects the error when progressing from time level t_n to t_{n-1} only. That is, we disregard the error already present in the solution at t_{n-1}. The real error at a point, however, depends on the error development over all previous time steps. This error, $e^n = u^n - u_{\mathrm{e}}(t_n)$, is known as the *global error*. We may look at u^n for some n and Taylor expand the mathematical expressions as functions of $p = a\Delta t$ to get a simple expression for the global error (for small p):

```
>>> n = Symbol('n')
>>> u_e = exp(-p*n)
>>> u_n = A**n
>>> FE = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
>>> BE = u_e.series(p, 0, 4) - u_n.subs(theta, 1).series(p, 0, 4)
>>> CN = u_e.series(p, 0, 4) - u_n.subs(theta, half).series(p, 0, 4)
```

```
>>> FE
(1/2)*n*p**2 - 1/2*n**2*p**3 + (1/3)*n*p**3 + O(p**4)
>>> BE
(1/2)*n**2*p**3 - 1/2*n*p**2 + (1/3)*n*p**3 + O(p**4)
>>> CN
(1/12)*n*p**3 + O(p**4)
```

Note that `sympy` does not sort the polynomial terms in the output, so p^3 appears before p^2 in the output of `BE`.

For a fixed time t, the parameter n in these expressions increases as $p \to 0$ since $t = n\Delta t = \text{const}$ and hence n must increase like Δt^{-1}. With n substituted by $t/\Delta t$ in the leading-order error terms, these become

$$e^n = \frac{1}{2}np^2 = \frac{1}{2}ta^2\Delta t, \qquad \text{Forward Euler} \tag{68}$$

$$e^n = -\frac{1}{2}np^2 = -\frac{1}{2}ta^2\Delta t, \qquad \text{Backward Euler} \tag{69}$$

$$e^n = \frac{1}{12}np^3 = \frac{1}{12}ta^3\Delta t^2, \qquad \text{Crank-Nicolson} \tag{70}$$

The global error is therefore of second order (in Δt) for the Crank-Nicolson scheme and of first order for the other two schemes.

Convergence.

When the global error $e^n \to 0$ as $\Delta t \to 0$, we say that the scheme is *convergent*. It means that the numerical solution approaches the exact solution as the mesh is refined, and this is a much desired property of a numerical method.

4.8 Integrated errors

It is common to study the norm of the numerical error, as explained in detail in Section 2.4. The L^2 norm can be computed by treating e^n as a function of t in `sympy` and performing symbolic integration. For the Forward Euler scheme we have

```
p, n, a, dt, t, T, theta = symbols('p n a dt t T 'theta')
A = (1-(1-theta)*p)/(1+theta*p)
u_e = exp(-p*n)
u_n = A**n
error = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
# Introduce t and dt instead of n and p
error = error.subs('n', 't/dt').subs(p, 'a*dt')
error = error.as_leading_term(dt) # study only the first term
print error
error_L2 = sqrt(integrate(error**2, (t, 0, T)))
print error_L2
```

The output reads

```
sqrt(30)*sqrt(T**3*a**4*dt**2*(6*T**2*a**2 - 15*T*a + 10))/60
```

which means that the L^2 error behaves like $a^2\Delta t$.

Strictly speaking, the numerical error is only defined at the mesh points so it makes most sense to compute the ℓ^2 error

$$||e^n||_{\ell^2} = \sqrt{\Delta t \sum_{n=0}^{N_t} (u_{\mathrm{e}}(t_n) - u^n)^2}\,.$$

We have obtained an exact analytical expression for the error at $t = t_n$, but here we use the leading-order error term only since we are mostly interested in how the error behaves as a polynomial in Δt, and then the leading order term will dominate. For the Forward Euler scheme, $u_{\mathrm{e}}(t_n) - u^n \approx \frac{1}{2}np^2$, and we have

$$||e^n||_{\ell^2}^2 = \Delta t \sum_{n=0}^{N_t} \frac{1}{4} n^2 p^4 = \Delta t \frac{1}{4} p^4 \sum_{n=0}^{N_t} n^2\,.$$

Now, $\sum_{n=0}^{N_t} n^2 \approx \frac{1}{3}N_t^3$. Using this approximation, setting $N_t = T/\Delta t$, and taking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{2}\sqrt{\frac{T^3}{3}}a^2\Delta t\,. \tag{71}$$

Calculations for the Backward Euler scheme are very similar and provide the same result, while the Crank-Nicolson scheme leads to

$$||e^n||_{\ell^2} = \frac{1}{12}\sqrt{\frac{T^3}{3}}a^3\Delta t^2\,. \tag{72}$$

Summary of errors.

Both the global point-wise errors (68)-(70) and their time-integrated versions (71) and (72) show that

- the Crank-Nicolson scheme is of second order in Δt, and
- the Forward Euler and Backward Euler schemes are of first order in Δt.

4.9 Truncation error

The truncation error is a very frequently used error measure for finite difference methods. It is defined as *the error in the difference equation that arises when inserting the exact solution.* Contrary to many other error measures, e.g., the true error $e^n = u_{\mathrm{e}}(t_n) - u^n$, the truncation error is a quantity that is easily computable.

Before reading on, it is wise to review Section 1.7 on how Taylor polynomials were used to derive finite differences and quantify the error in the formulas. Very similar reasoning, and almost identical mathematical details, will be carried out below, but in a slightly different context. Now, the focus is on the error when solving a differential equation, while in Section 1.7 we derived errors for a finite difference formula. These errors are tightly connected in the present model problem.

Let us illustrate the calculation of the truncation error for the Forward Euler scheme. We start with the difference equation on operator form,

$$[D_t^+ u = -au]^n,$$

which is the short form for

$$\frac{u^{n+1} - u^n}{\Delta t} = -au^n\,.$$

The idea is to see how well the exact solution $u_{\mathrm{e}}(t)$ fulfills this equation. Since $u_{\mathrm{e}}(t)$ in general will not obey the discrete equation, we get an error in the discrete equation. This error is called a *residual*, denoted here by R^n:

$$R^n = \frac{u_{\mathrm{e}}(t_{n+1}) - u_{\mathrm{e}}(t_n)}{\Delta t} + au_{\mathrm{e}}(t_n)\,. \tag{73}$$

The residual is defined at each mesh point and is therefore a mesh function with a superscript n.

The interesting feature of R^n is to see how it depends on the discretization parameter Δt. The tool for reaching this goal is to Taylor expand u_{e} around the point where the difference equation is supposed to hold, here $t = t_n$. We have that

$$u_{\mathrm{e}}(t_{n+1}) = u_{\mathrm{e}}(t_n) + u_{\mathrm{e}}'(t_n)\Delta t + \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t^2 + \cdots\,,$$

which may be used to reformulate the fraction in (73) so that

$$R^n = u_{\mathrm{e}}'(t_n) + \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t + \ldots + au_{\mathrm{e}}(t_n)\,.$$

Now, u_{e} fulfills the ODE $u_{\mathrm{e}}' = -au_{\mathrm{e}}$, which means that the first and last term cancel and we have

$$R^n = \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,.$$

This R^n is the *truncation error*, which for the Forward Euler is seen to be of first order in Δt as $\Delta \to 0$.

The above procedure can be repeated for the Backward Euler and the Crank-Nicolson schemes. We start with the scheme in operator notation, write it out in detail, Taylor expand u_{e} around the point $\tilde{t}$ at which the difference equation is defined, collect terms that correspond to the ODE (here $u'_{\mathrm{e}} + au_{\mathrm{e}}$), and identify the remaining terms as the residual R, which is the truncation error. The Backward Euler scheme leads to

$$R^n \approx -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t,$$

while the Crank-Nicolson scheme gives

$$R^{n+\frac{1}{2}} \approx \frac{1}{24}u'''_{\mathrm{e}}(t_{n+\frac{1}{2}})\Delta t^2,$$

when $\Delta t \to 0$.

The *order* r of a finite difference scheme is often defined through the leading term Δt^r in the truncation error. The above expressions point out that the Forward and Backward Euler schemes are of first order, while Crank-Nicolson is of second order. We have looked at other error measures in other sections, like the error in amplification factor and the error $e^n = u_{\mathrm{e}}(t_n) - u^n$, and expressed these error measures in terms of Δt to see the order of the method. Normally, calculating the truncation error is more straightforward than deriving the expressions for other error measures and therefore the easiest way to establish the order of a scheme.

4.10 Consistency, stability, and convergence

Three fundamental concepts when solving differential equations by numerical methods are consistency, stability, and convergence. We shall briefly touch upon these concepts below in the context of the present model problem.

Consistency means that the error in the difference equation, measured through the truncation error, goes to zero as $\Delta t \to 0$. Since the truncation error tells how well the exact solution fulfills the difference equation, and the exact solution fulfills the differential equation, consistency ensures that the difference equation approaches the differential equation in the limit. The expressions for the truncation errors in the previous section are all proportional to Δt or Δt^2, hence they vanish as $\Delta t \to 0$, and all the schemes are consistent. Lack of consistency implies that we actually solve some other differential equation in the limit $\Delta t \to 0$ than we aim at.

Stability means that the numerical solution exhibits the same qualitative properties as the exact solution. This is obviously a feature we want the numerical solution to have. In the present exponential decay model, the exact solution is monotone and decaying. An increasing numerical solution is not in accordance with the decaying nature of the exact solution and hence unstable. We can also say that an oscillating numerical solution lacks the property of monotonicity

of the exact solution and is also unstable. We have seen that the Backward Euler scheme always leads to monotone and decaying solutions, regardless of Δt, and is hence stable. The Forward Euler scheme can lead to increasing solutions and oscillating solutions if Δt is too large and is therefore unstable unless Δt is sufficiently small. The Crank-Nicolson can never lead to increasing solutions and has no problem to fulfill that stability property, but it can produce oscillating solutions and is unstable in that sense, unless Δt is sufficiently small.

Convergence implies that the global (true) error mesh function $e^n = u_{\mathrm{e}}(t_n) - u^n \to 0$ as $\Delta t \to 0$. This is really what we want: the numerical solution gets as close to the exact solution as we request by having a sufficiently fine mesh.

Convergence is hard to establish theoretically, except in quite simple problems like the present one. Stability and consistency are much easier to calculate. A major breakthrough in the understanding of numerical methods for differential equations came in 1956 when Lax and Richtmeyer established equivalence between convergence on one hand and consistency and stability on the other (the Lax equivalence theorem). In practice it meant that one can first establish that a method is stable and consistent, and then it is automatically convergent (which is much harder to establish). The result holds for linear problems only, and in the world of nonlinear differential equations the relations between consistency, stability, and convergence are much more complicated.

We have seen in the previous analysis that the Forward Euler, Backward Euler, and Crank-Nicolson schemes are convergent ($e^n \to 0$), that they are consistent ($R^n \to 0$, and that they are stable under certain conditions on the size of Δt. We have also derived explicit mathematical expressions for e^n, the truncation error, and the stability criteria.

5 Exercises

Exercise 7: Visualize the accuracy of finite differences

The purpose of this exercise is to visualize the accuracy of finite difference approximations of the derivative of a given function. For any finite difference approximation, take the Forward Euler difference as an example, and any specific function, take $u = e^{-at}$, we may introduce an error fraction

$$E = \frac{[D_t^+ u]^n}{u'(t_n)} = \frac{\exp\left(-a(t_n + \Delta t)\right) - \exp\left(-at_n\right)}{-a\exp\left(-at_n\right)\Delta t} = \frac{1}{a\Delta t}\left(1 - \exp\left(-a\Delta t\right)\right),$$

and view E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while E may deviate significantly from unity for large Δt. How the error depends on Δt is best visualized in a graph where we use a logarithmic scale for Δt, so we can cover many orders of magnitude of that quantity. Here is a code segment creating an array of 100 intervals, on the logarithmic scale, ranging from 10^{-6} to $10^{-0.5}$ and then plotting E versus $p = a\Delta t$ with logarithmic scale on the p axis:

```
from numpy import logspace, exp
from matplotlib.pyplot import semilogx
p = logspace(-6, -0.5, 101)
y = (1-exp(-p))/p
semilogx(p, y)
```

Illustrate such errors for the finite difference operators $[D_t^+ u]^n$ (forward), $[D_t^- u]^n$ (backward), and $[D_t u]^n$ (centered) in the same plot.

Perform a Taylor series expansions of the error fractions and find the leading order r in the expressions of type $1 + Cp^r + \mathcal{O}(p^{r+1})$, where C is some constant.

Hint. To save manual calculations and learn more about symbolic computing, make functions for the three difference operators and use `sympy` to perform the symbolic differences, differentiation, and Taylor series expansion. To plot a symbolic expression `E` against `p`, convert the expression to a Python function first: `E = sympy.lamdify([p], E)`.

Solution. Here is Python code for the exercise:

```
import sympy as sym

# Define finite difference operators as functions

def D_f(u, dt, t):
    return (u(t + dt) - u(t))/dt

def D_b(u, dt, t):
    return (u(t) - u(t - dt))/dt

def D_c(u, dt, t):
    return (u(t + dt) - u(t - dt))/(2*dt)


def make_plot():
    def u(t):
        return sym.exp(-a*t)

    a, t, dt, p = sym.symbols('a t dt p')
    dudt = sym.diff(u(t), t)

    from numpy import logspace, exp
    from matplotlib.pyplot import (
        semilogx, legend, show, loglog, savefig)

    # Map operator function name to logical names
    operator2name = dict(
        D_f='forward', D_b='backward', D_c='central')
    legends = []
    for operator in D_f, D_b, D_c:
        E = operator(u, dt, t)/dudt
        # Expand, set p=a*dt, simplify
        E = sym.expand(E)
        E = E.subs(a*dt, p)
        E = sym.simplify(E)
        print '%s E:' % operator2name[operator.__name__], E
```

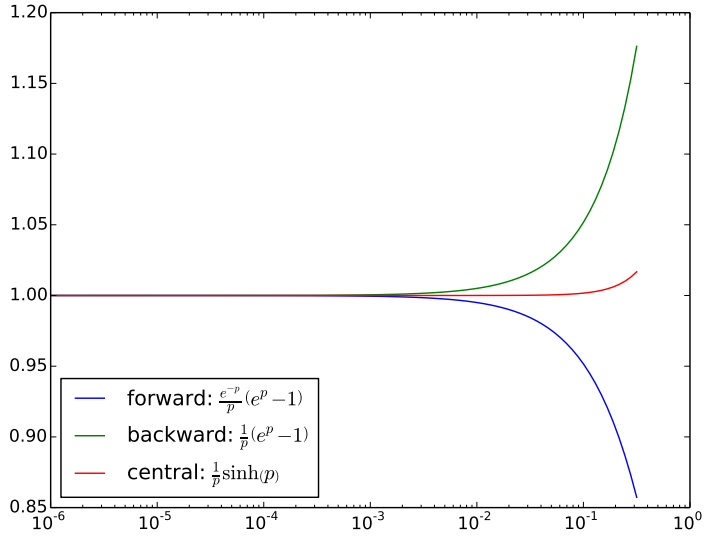


Figure 16: Plot for Exercise 7.

```
        print 'Taylor series:', E.series(p, 0, 3)
        latex_expr = sym.latex(E)

        E = sym.lambdify([p], E, modules='numpy')
        p_values = logspace(-6, -0.5, 101)
        y = E(p_values)
        semilogx(p_values, y)
        legends.append(operator2name[operator.__name__] +
                       ': $' + latex_expr + '$')
    legend(legends, loc='lower left')
    savefig('tmp.png'); savefig('tmp.pdf')
    show()

make_plot()
```

The output of the Taylor polynomials reads

```
forward E: (exp(p) - 1)*exp(-p)/p
Taylor series: 1 - p/2 + p**2/6 + O(p**3)
backward E: (exp(p) - 1)/p
Taylor series: 1 + p/2 + p**2/6 + O(p**3)
central E: sinh(p)/p
Taylor series: 1 + p**2/6 + O(p**3)
```

Filename: `decay_plot_fd_error`.

Exercise 8: Explore the θ-rule for exponential growth

This exercise asks you to solve the ODE $u' = -au$ with $a < 0$ such that the ODE models exponential growth instead of exponential decay. A central theme is to investigate numerical artifacts and non-physical solution behavior.

a) Set $a = -1$ and run experiments with $\theta = 0, 0.5, 1$ for various values of Δt to uncover numerical artifacts. Recall that the exact solution is a monotone, growing function when $a < 0$. Oscillations or significantly wrong growth are signs of wrong qualitative behavior.

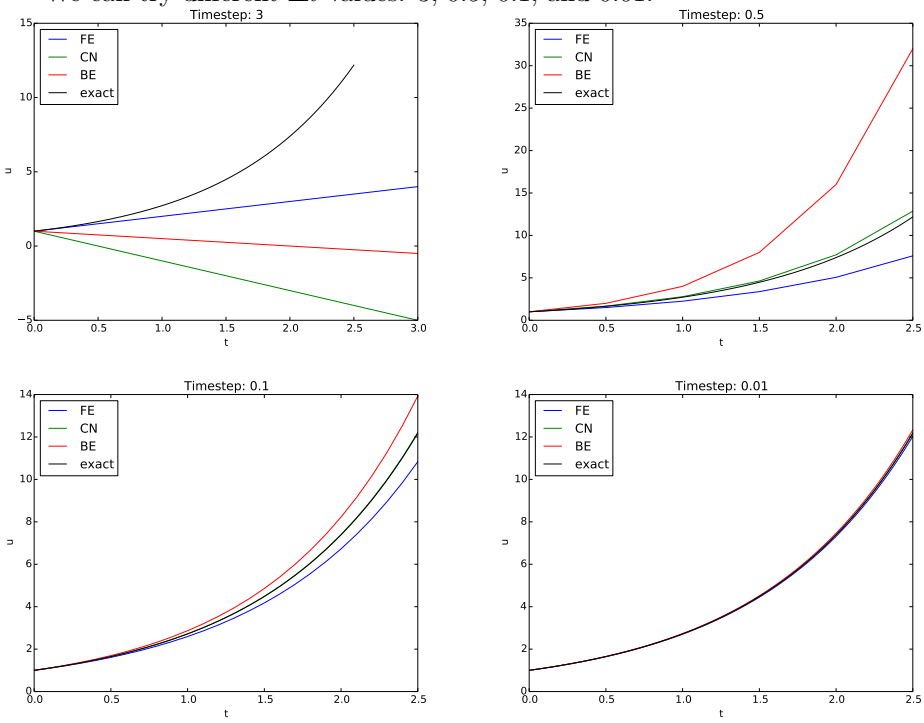
From the experiments, select four values of Δt that demonstrate the kind of numerical solutions that are characteristic for this model.

Solution. The schemes are exactly the same as in the case $a > 0$. A program solving the problem numerically is shown below.

```
from numpy import *

# Exercise a

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t

def exact_solution(t, I, a):
    return I*exp(-a*t)

def numerical_and_exact(theta, I, a, T, dt):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(I=I, a=a, T=T, dt=dt, theta=theta)

    t_e = linspace(0, T, 1001)        # fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    return u, t, u_e, t_e

def demo(dt):
    from matplotlib.pyplot import (
        plot, xlabel, ylabel, legend, title, savefig, show)
    for theta in [0, 0.5, 1]:
        u, t, u_e, t_e = numerical_and_exact(
            I=1, a=-1, T=2.5, dt=dt, theta=theta)
        xlabel('t')
        ylabel('u')
        plot(t,   u)

    plot(t_e, u_e, 'k-')  # black line
    legend(['FE', 'CN', 'BE', 'exact'], loc='upper left')
    title('Timestep: %g' % dt)
```

```
    savefig('tmp_%g.png' % dt); savefig('tmp_%g.pdf' % dt)
    show()
```

We can try different Δt values: 3, 0.5, 0.1, and 0.01.



b) Write up the amplification factor and plot it for $\theta = 0, 0.5, 1$ together with the exact one for $a\Delta t < 0$. Use the plot to explain the observations made in the experiments.

Hint. Modify the `decay_ampf_plot.py` code.

Solution. The amplification factor is the same as when $a > 0$, but here we introduce $p = -a\Delta t > 0$ since $a < 0$:

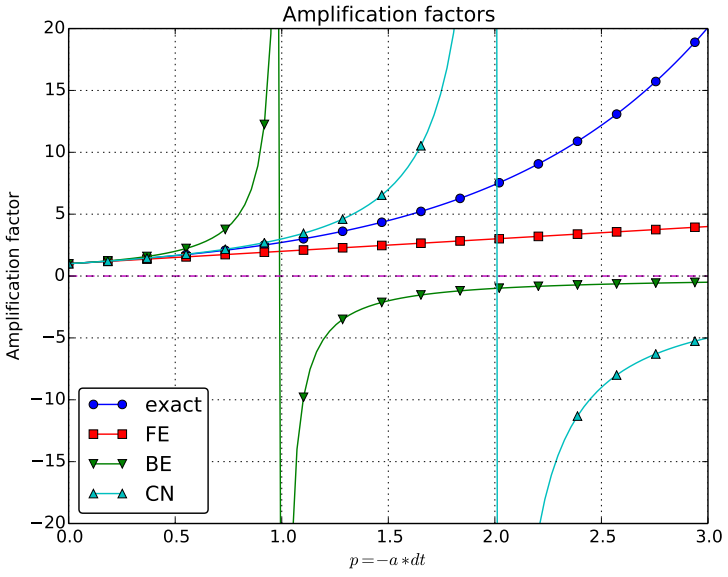
$$A(p) = \frac{1 + (1-\theta)p}{1 - \theta p}. \tag{74}$$

A major problem is that the denominator can be zero when $a < 0$. This happens for $p = 1/\theta$. The exact amplification factor is $A_{\mathrm{e}} = e^{p}$.

Here is code for computing and plotting the factors:

```
# Exercise b

def plot_amplification_factors(names):
    # Substitute -p by p since a is negative for a growth model
```

```
def A_exact(p):
    return exp(p)

def A(p, theta):
    return (1+(1-theta)*p)/(1-theta*p)

def amplification_factor(names):
    # Use SciTools since it adds markers to colored lines
    from scitools.std import (
        plot, title, xlabel, ylabel, hold, savefig,
        axis, legend, grid, show, figure)
    figure()
    curves = {}
    p = linspace(0, 3, 99)
    curves['exact'] = A_exact(p)
    plot(p, curves['exact'])
    hold('on')
    name2theta = dict(FE=0, BE=1, CN=0.5)
    for name in names:
        curves[name] = A(p, name2theta[name])
        plot(p, curves[name])
        axis([p[0], p[-1], -20, 20])
        #semilogy(p, curves[name])
    plot([p[0], p[-1]], [0, 0], '--')  # A=0 line
    title('Amplification factors')
    grid('on')
    legend(['exact'] + names, loc='lower left', fancybox=True)
    xlabel(r'$p=-a\cdot dt$')
    ylabel('Amplification factor')
    savefig('A_growth.png'); savefig('A_growth.pdf')
    #show()

amplification_factor(names)
```



Filename: `exponential_growth`.

6 Model extensions

It is time to consider generalizations of the simple decay model $u = -au$ and also to look at additional numerical solution methods.

6.1 Generalization: including a variable coefficient

In the ODE for decay, $u' = -au$, we now consider the case where a depends on time:

$$u'(t) = -a(t)u(t), \quad t \in (0,T], \quad u(0) = I\,. \tag{75}$$

A Forward Euler scheme consist of evaluating (75) at $t = t_n$ and approximating the derivative with a forward difference $[D_t^+ u]^n$:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_n)u^n\,. \tag{76}$$

The Backward Euler scheme becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)u^n\,. \tag{77}$$

The Crank-Nicolson method builds on sampling the ODE at $t_{n+\frac{1}{2}}$. We can evaluate a at $t_{n+\frac{1}{2}}$ and use an average for u at times t_n and t_{n+1}:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_{n+\frac{1}{2}})\frac{1}{2}(u^n + u^{n+1})\,. \tag{78}$$

Alternatively, we can use an average for the product au:

$$\frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{2}(a(t_n)u^n + a(t_{n+1})u^{n+1})\,. \tag{79}$$

The θ-rule unifies the three mentioned schemes. One version is to have a evaluated at the weighted time point $(1-\theta)t_n + \theta t_{n+1}$,

$$\frac{u^{n+1} - u^n}{\Delta t} = -a((1-\theta)t_n + \theta t_{n+1})((1-\theta)u^n + \theta u^{n+1})\,. \tag{80}$$

Another possibility is to apply a weighted average for the product au,

$$\frac{u^{n+1} - u^n}{\Delta t} = -(1-\theta)a(t_n)u^n - \theta a(t_{n+1})u^{n+1}\,. \tag{81}$$

With the finite difference operator notation the Forward Euler and Backward Euler schemes can be summarized as

$$[D_t^+ u = -au]^n, \tag{82}$$
$$[D_t^- u = -au]^n\,. \tag{83}$$

The Crank-Nicolson and θ schemes depend on whether we evaluate a at the sample point for the ODE or if we use an average. The various versions are written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}, \tag{84}$$
$$[D_t u = -\overline{au}^t]^{n+\frac{1}{2}}, \tag{85}$$
$$[D_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \tag{86}$$
$$[D_t u = -\overline{au}^{t,\theta}]^{n+\theta}\,. \tag{87}$$

6.2 Generalization: including a source term

A further extension of the model ODE is to include a source term $b(t)$:

$$u'(t) = -a(t)u(t) + b(t), \quad t \in (0,T], \quad u(0) = I\,. \tag{88}$$

The time point where we sample the ODE determines where $b(t)$ is evaluated. For the Crank-Nicolson scheme and the θ-rule we have a choice of whether to evaluate $a(t)$ and $b(t)$ at the correct point or use an average. The chosen strategy becomes particularly clear if we write up the schemes in the operator notation:

$$[D_t^+ u = -au + b]^n, \tag{89}$$
$$[D_t^- u = -au + b]^n, \tag{90}$$
$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{91}$$
$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}}, \tag{92}$$
$$[D_t u = -a\overline{u}^{t,\theta} + b]^{n+\theta}, \tag{93}$$
$$[D_t u = \overline{-au + b}^{t,\theta}]^{n+\theta}\,. \tag{94}$$

6.3 Implementation of the generalized model problem

Deriving the θ-rule formula. Writing out the θ-rule in (94), using (44) and (45), we get

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta(-a^{n+1}u^{n+1} + b^{n+1})) + (1-\theta)(-a^n u^n + b^n)), \tag{95}$$

where a^n means evaluating a at $t = t_n$ and similar for a^{n+1}, b^n, and b^{n+1}. We solve for u^{n+1}:

$$u^{n+1} = ((1 - \Delta t(1-\theta)a^n)u^n + \Delta t(\theta b^{n+1} + (1-\theta)b^n))(1 + \Delta t\theta a^{n+1})^{-1}\,. \tag{96}$$

Python code. Here is a suitable implementation of (95) where $a(t)$ and $b(t)$ are given as Python functions:

```
def solver(I, a, b, T, dt, theta):
    """
    Solve u'=-a(t)*u + b(t), u(0)=I,
    for t in (0,T] with steps of dt.
    a and b are Python functions of t.
    """
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = ((1 - dt*(1-theta)*a(t[n]))*u[n] + \
                  dt*(theta*b(t[n+1]) + (1-theta)*b(t[n])))/\
                  (1 + dt*theta*a(t[n+1]))
    return u, t
```

This function is found in the file `decay_vc.py` (`vc` stands for "variable coefficients").

Coding of variable coefficients. The `solver` function shown above demands the arguments `a` and `b` to be Python functions of time `t`, say

```
def a(t):
    return a_0 if t < tp else k*a_0

def b(t):
    return 1
```

Here, `a(t)` has three parameters `a0`, `tp`, and `k`, which must be global variables.

A better implementation, which avoids global variables, is to represent `a` by a class where the parameters are attributes and where a *special method* `__call__` evaluates $a(t)$:

```
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k

    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0

a = A(a0=2, k=1)  # a behaves as a function a(t)
```

For quick tests it is cumbersome to write a complete function or a class. The *lambda function* construction in Python is then convenient. For example,

```
a = lambda t: a_0 if t < tp else k*a_0
```

is equivalent to the `def a(t)` definition above. In general,

```
f = lambda arg1, arg2, ...: expression
```

is equivalent to

```
def f(arg1, arg2, ...):
    return expression
```

One can use lambda functions directly in calls. Say we want to solve $u' = -u+1$, $u(0) = 2$:

```
u, t = solver(2, lambda t: 1, lambda t: 1, T, dt, theta)
```

Whether to use a plain function, a class, or a lambda function depends on the programmer's taste. Lazy programmers prefer the lambda construct, while very safe programmers go for the class solution.

6.4 Verifying a constant solution

An extremely useful partial verification method is to construct a test problem with a very simple solution, usually $u = \text{const}$. Especially the initial debugging of a program code can benefit greatly from such tests, because 1) all relevant numerical methods will exactly reproduce a constant solution, 2) many of the intermediate calculations are easy to control by hand for a constant u, and 3) even a constant u can uncover many bugs in an implementation.

The only constant solution for the problem $u' = -au$ is $u = 0$, but too many bugs can escape from that trivial solution. It is much better to search for a problem where $u = C = \text{const} \neq 0$. Then $u' = -a(t)u + b(t)$ is more appropriate: with $u = C$ we can choose any $a(t)$ and set $b = a(t)C$ and $I = C$. An appropriate test function is

```
def test_constant_solution():
    """
    Test problem where u=u_const is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return u_const

    def a(t):
        return 2.5*(1+t**3)  # can be arbitrary

    def b(t):
        return a(t)*u_const

    u_const = 2.15
    theta = 0.4; I = u_const; dt = 4
    Nt = 4  # enough with a few steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    print u
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    tol = 1E-14
    assert difference < tol
```

An interesting question is what type of bugs that will make the computed u^n deviate from the exact solution C. Fortunately, the updating formula and the initial condition must be absolutely correct for the test to pass! Any attempt to make a wrong indexing in terms like `a(t[n])` or any attempt to introduce an erroneous factor in the formula creates a solution that is different from C.

6.5 Verification via manufactured solutions

Following the idea of the previous section, we can choose any formula as the exact solution, insert the formula in the ODE problem and fit the data $a(t)$, $b(t)$, and I to make the chosen formula fulfill the equation. This powerful technique for generating exact solutions is very useful for verification purposes and known as the *method of manufactured solutions*, often abbreviated MMS.

One common choice of solution is a linear function in the independent variable(s). The rationale behind such a simple variation is that almost any relevant numerical solution method for differential equation problems is able to reproduce a linear function exactly to machine precision (if u is about unity in size; precision is lost if u takes on large values, see Exercise 9). The linear solution also makes some stronger demands to the numerical method and the implementation than the constant solution used in Section 6.4, at least in more complicated applications. Still, the constant solution is often ideal for initial debugging before proceeding with a linear solution.

We choose a linear solution $u(t) = ct + d$. From the initial condition it follows that $d = I$. Inserting this u in the left-hand side of (88), i.e., the ODE, we get

$$c = -a(t)u + b(t)\,.$$

Any function $u = ct + I$ is then a correct solution if we choose

$$b(t) = c + a(t)(ct + I)\,.$$

With this $b(t)$ there are no restrictions on $a(t)$ and c.

Let us prove that such a linear solution obeys the numerical schemes. To this end, we must check that $u^n = ca(t_n)(ct_n + I)$ fulfills the discrete equations. For these calculations, and later calculations involving linear solutions inserted in finite difference schemes, it is convenient to compute the action of a difference operator on a linear function t:

$$[D_t^+ t]^n = \frac{t_{n+1} - t_n}{\Delta t} = 1, \tag{97}$$

$$[D_t^- t]^n = \frac{t_n - t_{n-1}}{\Delta t} = 1, \tag{98}$$

$$[D_t t]^n = \frac{t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}}}{\Delta t} = \frac{(n+\frac{1}{2})\Delta t - (n-\frac{1}{2})\Delta t}{\Delta t} = 1\,. \tag{99}$$

Clearly, all three finite difference approximations to the derivative are exact for $u(t) = t$ or its mesh function counterpart $u^n = t_n$.

The difference equation for the Forward Euler scheme

$$[D_t^+ u = -au + b]^n,$$

with $a^n = a(t_n)$, $b^n = c + a(t_n)(ct_n + I)$, and $u^n = ct_n + I$ then results in

$$c = -a(t_n)(ct_n + I) + c + a(t_n)(ct_n + I) = c$$

which is always fulfilled. Similar calculations can be done for the Backward Euler and Crank-Nicolson schemes, or the θ-rule for that matter. In all cases, $u^n = ct_n + I$ is an exact solution of the discrete equations. That is why we should expect that $u^n - u_{\mathrm{e}}(t_n) = 0$ mathematically and $|u^n - u_{\mathrm{e}}(t_n)|$ less than a small number about the machine precision for $n = 0, \ldots, N_t$.

The following function offers an implementation of this verification test based on a linear exact solution:

```
def test_linear_solution():
    """
    Test problem where u=c*t+I is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return c*t + I

    def a(t):
        return t**0.5  # can be arbitrary

    def b(t):
        return c + a(t)*exact_solution(t)

    theta = 0.4; I = 0.1; dt = 0.1; c = -0.5
    T = 4
    Nt = int(T/dt)  # no of steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    print difference
    tol = 1E-14  # depends on c!
    assert difference < tol
```

Any error in the updating formula makes this test fail!

Choosing more complicated formulas as the exact solution, say $\cos(t)$, will not make the numerical and exact solution coincide to machine precision, because finite differencing of $\cos(t)$ does not exactly yield the exact derivative $-\sin(t)$. In such cases, the verification procedure must be based on measuring the convergence rates as exemplified in Section 6.6. Convergence rates can be computed as long as one has an exact solution of a problem that the solver can be tested on, but this can always be obtained by the method of manufactured solutions.

6.6 Computing convergence rates

We expect that the error E in the numerical solution is reduced if the mesh size Δt is decreased. More specifically, many numerical methods obey a power-law relation between E and Δt:

$$E = C\Delta t^r, \tag{100}$$

where C and r are (usually unknown) constants independent of Δt. The formula (100) is viewed as an asymptotic model valid for sufficiently small Δt. How small is normally hard to estimate without doing numerical estimations of r.

The parameter r is known as the *convergence rate*. For example, if the convergence rate is 2, halving Δt reduces the error by a factor of 4. Diminishing Δt then has a greater impact on the error compared with methods that have $r = 1$. For a given value of r, we refer to the method as of r-th order. First- and second-order methods are most common in scientific computing.

Estimating r. There are two alternative ways of estimating C and r based on a set of m simulations with corresponding pairs $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$, and $\Delta t_i < \Delta t_{i-1}$ (i.e., decreasing cell size).

1. Take the logarithm of (100), $\ln E = r \ln \Delta t + \ln C$, and fit a straight line to the data points $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$.
2. Consider two consecutive experiments, $(\Delta t_i, E_i)$ and $(\Delta t_{i-1}, E_{i-1})$. Dividing the equation $E_{i-1} = C\Delta t_{i-1}^r$ by $E_i = C\Delta t_i^r$ and solving for r yields

$$r_{i-1} = \frac{\ln(E_{i-1}/E_i)}{\ln(\Delta t_{i-1}/\Delta t_i)} \tag{101}$$

for $i = 1, \ldots, m-1$. Note that we have introduced a subindex $i-1$ on r in (101) because r estimated from a pair of experiments must be expected to change with i.

The disadvantage of method 1 is that (100) might not be valid for the coarsest meshes (largest Δt values). Fitting a line to all the data points is then misleading. Method 2 computes convergence rates for pairs of experiments and allows us to see if the sequence r_i converges to some value as $i \to m-2$. The final r_{m-2} can then be taken as the convergence rate. If the coarsest meshes have a differing rate, the corresponding time steps are probably too large for (100) to be valid. That is, those time steps lie outside the asymptotic range of Δt values where the error behaves like (100).

Implementation. Suppose we have some function `error(dt, ...)` that can compute the numerical error in our mathematical model. To compute $r_0, r_1, \ldots, r_{m-2}$ from (100), we just wrap a loop over Δt values (`dt_values`) around the `error` function:

```
E_values = [error(dt, ...) for dt in dt_values]

# Compute pairwise convergence rates
m = len(dt_values)
r = [log(E_values[i-1]/E_values[i])/
     log(dt_values[i-1]/dt_values[i])
```

```
     for i in range(1, m, 1)]

# Strip off to 2 decimals
r = [round(r_, 2) for r_ in r]
```

We can run the convergence rate estimate computations for the θ-rule discretization of $u' = -au$, using $\Delta t = 0.5, -.25, 0.1, 0.05, 0.025, 0.01$:

```
FE: 1.33 1.15 1.07 1.03 1.02
BE: 0.98 0.99 0.99 1.00 1.00
CN: 2.14 2.07 2.03 2.01 2.01
```

The Forward and Backward Euler methods seem to have an r value which stabilizes at 1, while the Crank-Nicolson seems to be a second-order method with $r = 2$. These results are in very good agreement with various theoretical considerations for r.

Why convergence rates are important.

The strong practical application of computing convergence rates is for verification: wrong convergence rates point to errors in the code, and correct convergence rates bring strong support for a correct implementation. Experience shows that bugs in the code easily destroy the expected convergence rate.

6.7 Extension to systems of ODEs

Many ODE models involve more than one unknown function and more than one equation. Here is an example of two unknown functions $u(t)$ and $v(t)$:

$$u' = au + bv, \tag{102}$$
$$v' = cu + dv, \tag{103}$$

for constants a, b, c, d. Applying the Forward Euler method to each equation results in a simple updating formula:

$$u^{n+1} = u^n + \Delta t(au^n + bv^n), \tag{104}$$
$$v^{n+1} = u^n + \Delta t(cu^n + dv^n)\,. \tag{105}$$

On the other hand, the Crank-Nicolson or Backward Euler schemes result in a 2×2 linear system for the new unknowns. The latter scheme becomes

$$u^{n+1} = u^n + \Delta t(au^{n+1} + bv^{n+1}), \tag{106}$$
$$v^{n+1} = v^n + \Delta t(cu^{n+1} + dv^{n+1})\,. \tag{107}$$

Collecting u^{n+1} as well as v^{n+1} on the left-hand side results in

$$(1 - \Delta t a)u^{n+1} + bv^{n+1} = u^n, \tag{108}$$

$$cu^{n+1} + (1 - \Delta t d)v^{n+1} = v^n, \tag{109}$$

which is a system of two coupled, linear, algebraic equations in two unknowns. These equations can be solved by hand (using standard techniques for two algebraic equations with two unknowns x and y), resulting in explicit formulas for u^{n+1} and v^{n+1} that can be directly implemented. For systems of ODEs with many equations and unknowns, one will express the coupled equations at each time level in matrix form and call software for numerical solution of linear systems of equations.

7 General first-order ODEs

We now turn the attention to general, nonlinear ODEs and systems of such ODEs. Our focus is on numerical methods that can be readily reused for time-discretization of PDEs, and diffusion PDEs in particular. The methods are just briefly listed, and we refer to the rich literature for more detailed descriptions and analysis - the books [10, 1, 2, 3] are all excellent resources on numerical methods for ODEs. We also demonstrate the Odespy Python interface to a range of different software for general first-order ODE systems.

7.1 Generic form of first-order ODEs

ODEs are commonly written in the generic form

$$u' = f(u, t), \quad u(0) = I, \tag{110}$$

where $f(u, t)$ is some prescribed function. As an example, our most general exponential decay model (88) has $f(u, t) = -a(t)u(t) + b(t)$.

The unknown u in (110) may either be a scalar function of time t, or a vector valued function of t in case of a *system of ODEs* with m unknown components:

$$u(t) = (u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t)) \,.$$

In that case, the right-hand side is vector-valued function with m components,

$$\begin{aligned} f(u, t) = (&f^{(0)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)), \\ &f^{(1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)), \\ &\vdots, \\ &f^{(m-1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t))) \,. \end{aligned}$$

Actually, any system of ODEs can be written in the form (110), but higher-order ODEs then need auxiliary unknown functions to enable conversion to a first-order system.

Next we list some well-known methods for $u' = f(u,t)$, valid both for a single ODE (scalar u) and systems of ODEs (vector u).

7.2 The θ-rule

The θ-rule scheme applied to $u' = f(u,t)$ becomes

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta f(u^{n+1}, t_{n+1}) + (1-\theta) f(u^n, t_n)\,. \tag{111}$$

Bringing the unknown u^{n+1} to the left-hand side and the known terms on the right-hand side gives

$$u^{n+1} - \Delta t\theta f(u^{n+1}, t_{n+1}) = u^n + \Delta t(1-\theta) f(u^n, t_n)\,. \tag{112}$$

For a general f (not linear in u), this equation is *nonlinear* in the unknown u^{n+1} unless $\theta = 0$. For a scalar ODE ($m = 1$), we have to solve a single nonlinear algebraic equation for u^{n+1}, while for a system of ODEs, we get a system of coupled, nonlinear algebraic equations. Newton's method is a popular solution approach in both cases. Note that with the Forward Euler scheme ($\theta = 0$) we do not have to deal with nonlinear equations, because in that case we have an explicit updating formula for u^{n+1}. This is known as an *explicit* scheme. With $\theta \neq 1$ we have to solve (systems of) algebraic equations, and the scheme is said to be *implicit*.

7.3 An implicit 2-step backward scheme

The implicit backward method with 2 steps applies a three-level backward difference as approximation to $u'(t)$,

$$u'(t_{n+1}) \approx \frac{3u^{n+1} - 4u^n + u^{n-1}}{2\Delta t},$$

which is an approximation of order Δt^2 to the first derivative. The resulting scheme for $u' = f(u,t)$ reads

$$u^{n+1} = \frac{4}{3}u^n - \frac{1}{3}u^{n-1} + \frac{2}{3}\Delta t f(u^{n+1}, t_{n+1})\,. \tag{113}$$

Higher-order versions of the scheme (113) can be constructed by including more time levels. These schemes are known as the Backward Differentiation Formulas (BDF), and the particular version (113) is often referred to as BDF2.

Note that the scheme (113) is implicit and requires solution of nonlinear equations when f is nonlinear in u. The standard 1st-order Backward Euler method or the Crank-Nicolson scheme can be used for the first step.

7.4 Leapfrog schemes

The ordinary Leapfrog scheme. The derivative of u at some point t_n can be approximated by a central difference over two time steps,

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n \tag{114}$$

which is an approximation of second order in Δt. The scheme can then be written as

$$[D_{2t}u = f(u,t)]^n,$$

in operator notation. Solving for u^{n+1} gives

$$u^{n+1} = u^{n-1} + 2\Delta t f(u^n, t_n)\,. \tag{115}$$

Observe that (115) is an explicit scheme, and that a nonlinear f (in u) is trivial to handle since it only involves the known u^n value. Some other scheme must be used as starter to compute u^1, preferably the Forward Euler scheme since it is also explicit.

The filtered Leapfrog scheme. Unfortunately, the Leapfrog scheme (115) will develop growing oscillations with time (see Problem 14). A remedy for such undesired oscillations is to introduce a *filtering technique*. First, a standard Leapfrog step is taken, according to (115), and then the previous u^n value is adjusted according to

$$u^n \;\leftarrow\; u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})\,. \tag{116}$$

The γ-terms will effectively damp oscillations in the solution, especially those with short wavelength (like point-to-point oscillations). A common choice of γ is 0.6 (a value used in the famous NCAR Climate Model).

7.5 The 2nd-order Runge-Kutta method

The two-step scheme

$$u^* = u^n + \Delta t f(u^n, t_n), \tag{117}$$

$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(f(u^n, t_n) + f(u^*, t_{n+1})\right), \tag{118}$$

essentially applies a Crank-Nicolson method (118) to the ODE, but replaces the term $f(u^{n+1}, t_{n+1})$ by a prediction $f(u^*, t_{n+1})$ based on a Forward Euler step (117). The scheme (117)-(118) is known as Huen's method, but is also a 2nd-order Runge-Kutta method. The scheme is explicit, and the error is expected to behave as Δt^2.

7.6 A 2nd-order Taylor-series method

One way to compute u^{n+1} given u^n is to use a Taylor polynomial. We may write up a polynomial of 2nd degree:

$$u^{n+1} = u^n + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 .$$

From the equation $u' = f(u,t)$ it follows that the derivatives of u can be expressed in terms of f and its derivatives:

$$\begin{aligned} u'(t_n) &= f(u^n, t_n), \\ u''(t_n) &= \frac{\partial f}{\partial u}(u^n, t_n)u'(t_n) + \frac{\partial f}{\partial t} \\ &= f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}, \end{aligned}$$

resulting in the scheme

$$u^{n+1} = u^n + f(u^n, t_n)\Delta t + \frac{1}{2}\left(f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}\right)\Delta t^2 . \tag{119}$$

More terms in the series could be included in the Taylor polynomial to obtain methods of higher order than 2.

7.7 The 2nd- and 3rd-order Adams-Bashforth schemes

The following method is known as the 2nd-order Adams-Bashforth scheme:

$$u^{n+1} = u^n + \frac{1}{2}\Delta t\left(3f(u^n, t_n) - f(u^{n-1}, t_{n-1})\right) . \tag{120}$$

The scheme is explicit and requires another one-step scheme to compute u^1 (the Forward Euler scheme or Heun's method, for instance). As the name implies, the error behaves like Δt^2.

Another explicit scheme, involving four time levels, is the 3rd-order Adams-Bashforth scheme

$$u^{n+1} = u^n + \frac{1}{12}\left(23f(u^n, t_n) - 16f(u^{n-1}, t_{n-1}) + 5f(u^{n-2}, t_{n-2})\right) . \tag{121}$$

The numerical error is of order Δt^3, and the scheme needs some method for computing u^1 and u^2.

More general, higher-order Adams-Bashforth schemes (also called *explicit Adams methods*) compute u^{n+1} as a linear combination of f at $k+1$ previous time steps:

$$u^{n+1} = u^n + \sum_{j=0}^{k} \beta_j f(u^{n-j}, t_{n-j}),$$

where β_j are known coefficients.

7.8 The 4th-order Runge-Kutta method

The perhaps most widely used method to solve ODEs is the 4th-order Runge-Kutta method, often called RK4. Its derivation is a nice illustration of common numerical approximation strategies, so let us go through the steps in detail to learn about algorithmic development.

The starting point is to integrate the ODE $u' = f(u,t)$ from t_n to t_{n+1}:

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u(t),t)dt\,.$$

We want to compute $u(t_{n+1})$ and regard $u(t_n)$ as known. The task is to find good approximations for the integral, since the integrand involves the unknown u between t_n and t_{n+1}.

The integral can be approximated by the famous Simpson's rule:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+\frac{1}{2}} + f^{n+1}\right).$$

The problem now is that we do not know $f^{n+\frac{1}{2}} = f(u^{n+\frac{1}{2}}, t_{n+\frac{1}{2}})$ and $f^{n+1} = (u^{n+1}, t_{n+1})$ as we know only u^n and hence f^n. The idea is to use various approximations for $f^{n+\frac{1}{2}}$ and f^{n+1} based on well-known schemes for the ODE in the intervals $[t_n, t_{n+\frac{1}{2}}]$ and $[t_n, t_{n+1}]$. We split the integral approximation into four terms:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

where $\hat{f}^{n+\frac{1}{2}}$, $\tilde{f}^{n+\frac{1}{2}}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+\frac{1}{2}}$ and f^{n+1}, respectively, that can be based on already computed quantities. For $\hat{f}^{n+\frac{1}{2}}$ we can apply an approximation to $u^{n+\frac{1}{2}}$ using the Forward Euler method with step $\frac{1}{2}\Delta t$:

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+\frac{1}{2}}) \tag{122}$$

Since this gives us a prediction of $f^{n+\frac{1}{2}}$, we can for $\tilde{f}^{n+\frac{1}{2}}$ try a Backward Euler method to approximate $u^{n+\frac{1}{2}}$:

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}})\,. \tag{123}$$

With $\tilde{f}^{n+\frac{1}{2}}$ as a hopefully good approximation to $f^{n+\frac{1}{2}}$, we can for the final term $\bar{f}^{n+1}$ use a Crank-Nicolson method to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1})\,. \tag{124}$$

We have now used the Forward and Backward Euler methods as well as the Crank-Nicolson method in the context of Simpson's rule. The hope is that the combination of these methods yields an overall time-stepping scheme from t_n to t_n+1 that is much more accurate than the $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta t^2)$ of the individual steps. This is indeed true: the overall accuracy is $\mathcal{O}(\Delta t^4)$!

To summarize, the 4th-order Runge-Kutta method becomes

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right), \tag{125}$$

where the quantities on the right-hand side are computed from (122)-(124). Note that the scheme is fully explicit so there is never any need to solve linear or nonlinear algebraic equations. However, the stability is conditional and depends on f. There is a whole range of *implicit* Runge-Kutta methods that are unconditionally stable, but require solution of algebraic equations involving f at each time step.

The simplest way to explore more sophisticated methods for ODEs is to apply one of the many high-quality software packages that exist, as the next section explains.

7.9 The Odespy software

A wide range of methods and software exist for solving (110). Many of the methods are accessible through a unified Python interface offered by the Odespy [8] package. Odespy features simple Python implementations of the most fundamental schemes as well as Python interfaces to several famous packages for solving ODEs: ODEPACK, Vode, rkc.f, rkf45.f, as well as the ODE solvers in SciPy, SymPy, and odelab.

The usage of Odespy follows this setup for the ODE $u' = -au$, $u(0) = I$, $t \in (0, T]$, here solved by the famous 4th-order Runge-Kutta method with $\Delta t = 1$ and $N_t = 6$ steps:

```
def f(u, t):
    return -a*u

import odespy
import numpy as np

I = 1; a = 0.5; Nt = 6; dt = 1
solver = odespy.RK4(f)
solver.set_initial_condition(I)
t_mesh = np.linspace(0, Nt*dt, Nt+1)
u, t = solver.solve(t_mesh)
```

The previously listed methods for ODEs are all accessible in Odespy:

- the θ-rule: `ThetaRule`
- special cases of the θ-rule: `ForwardEuler`, `BackwardEuler`, `CrankNicolson`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`

- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode`, `Lsoda`, `lsoda_scipy`
- The Leapfrog schemes: `Leapfrog` and `LeapfrogFiltered`

7.10 Example: Runge-Kutta methods

Since all solvers have the same interface in Odespy, except for a potentially different set of parameters in the solvers' constructors, one can easily make a list of solver objects and run a loop for comparing a lot of solvers. The code below, found in complete form in `decay_odespy.py`, compares the famous Runge-Kutta methods of orders 2, 3, and 4 with the exact solution of the decay equation $u' = -au$. Since we have quite long time steps, we have included the only relevant θ-rule for large time steps, the Backward Euler scheme ($\theta = 1$), as well. Figure 17 shows the results.

```
import numpy as np
import matplotlib.pyplot as plt
import sys

def f(u, t):
    return -a*u

I = 1; a = 2; T = 6
dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.75
Nt = int(round(T/dt))
t = np.linspace(0, Nt*dt, Nt+1)

solvers = [odespy.RK2(f),
           odespy.RK3(f),
           odespy.RK4(f),]

# BackwardEuler must use Newton solver to converge
# (Picard is default and leads to divergence)
solvers.append(
    odespy.BackwardEuler(f, nonlinear_solver='Newton'))
# Or tell BackwardEuler that it is a linear problem
solvers[-1] = odespy.BackwardEuler(f, f_is_linear=True,
                                   jac=lambda u, t: -a)]
legends = []
for solver in solvers:
    solver.set_initial_condition(I)
    u, t = solver.solve(t)

    plt.plot(t, u)
    plt.hold('on')
    legends.append(solver.__class__.__name__)

# Compare with exact solution plotted on a very fine mesh
t_fine = np.linspace(0, T, 10001)
u_e = I*np.exp(-a*t_fine)
plt.plot(t_fine, u_e, '-') # avoid markers by specifying line type
legends.append('exact')

plt.legend(legends)
```

```
plt.title('Time step: %g' % dt)
plt.show()
```

With the `odespy.BackwardEuler` method we must either tell that the problem is linear and provide the Jacobian of $f(u,t)$, i.e., $\partial f/\partial u$, as the `jac` argument, or we have to assume that f is nonlinear, but then specify Newton's method as solver for the nonlinear equations (since the equations are linear, Newton's method will converge in one iteration). By default, `odespy.BackwardEuler` assumes a nonlinear problem to be solved by Picard iteration, but that leads to divergence in the present problem.

Visualization tip.

We use Matplotlib for plotting here, but one could alternatively import `scitools.std` as `plt` instead. Plain use of Matplotlib as done here results in curves with different colors, which may be hard to distinguish on black-and-white paper. Using `scitools.std`, curves are automatically given colors *and* markers, thus making curves easy to distinguish on screen with colors and on black-and-white paper. The automatic adding of markers is normally a bad idea for a very fine mesh since all the markers get cluttered, but `scitools.std` limits the number of markers in such cases. For the exact solution we use a very fine mesh, but in the code above we specify the line type as a solid line (`-`), which means no markers and just a color to be automatically determined by the backend used for plotting (Matplotlib by default, but `scitools.std` gives the opportunity to use other backends to produce the plot, e.g., Gnuplot or Grace).

Also note the that the legends are based on the class names of the solvers, and in Python the name of the class type (as a string) of an object `obj` is obtained by `obj.__class__.__name__`.

The runs in Figure 17 and other experiments reveal that the 2nd-order Runge-Kutta method (`RK2`) is unstable for $\Delta t > 1$ and decays slower than the Backward Euler scheme for large and moderate Δt (see Exercise 13 for an analysis). However, for fine $\Delta t = 0.25$ the 2nd-order Runge-Kutta method approaches the exact solution faster than the Backward Euler scheme. That is, the latter scheme does a better job for larger Δt, while the higher order scheme is superior for smaller Δt. This is a typical trend also for most schemes for ordinary and partial differential equations.

The 3rd-order Runge-Kutta method (`RK3`) also has artifacts in the form of oscillatory behavior for the larger Δt values, much like that of the Crank-Nicolson scheme. For finer Δt, the 3rd-order Runge-Kutta method converges quickly to the exact solution.

The 4th-order Runge-Kutta method (`RK4`) is slightly inferior to the Backward Euler scheme on the coarsest mesh, but is then clearly superior to all the other schemes. It is definitely the method of choice for all the tested schemes.

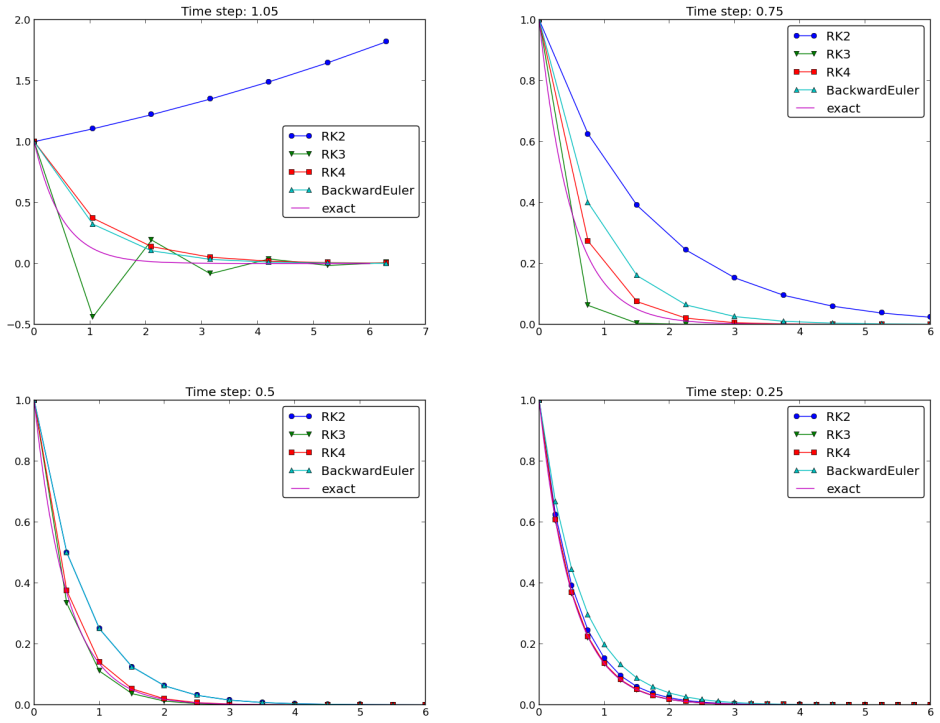


Figure 17: Behavior of different schemes for the decay equation.

Remark about using the θ-rule in Odespy. The Odespy package assumes that the ODE is written as $u' = f(u,t)$ with an f that is possibly nonlinear in u. The θ-rule for $u' = f(u,t)$ leads to

$$u^{n+1} = u^n + \Delta t \left(\theta f(u^{n+1}, t_{n+1}) + (1-\theta) f(u^n, t_n)\right),$$

which is a *nonlinear equation* in u^{n+1}. Odespy's implementation of the θ-rule (`ThetaRule`) and the specialized Backward Euler (`BackwardEuler`) and Crank-Nicolson (`CrankNicolson`) schemes must invoke iterative methods for solving the nonlinear equation in u^{n+1}. This is done even when f is linear in u, as in the model problem $u' = -au$, where we can easily solve for u^{n+1} by hand. Therefore, we need to specify use of Newton's method to solve the equations. (Odespy allows other methods than Newton's to be used, for instance Picard iteration, but that method is not suitable. The reason is that it applies the Forward Euler scheme to generate a start value for the iterations. Forward Euler may give very wrong solutions for large Δt values. Newton's method, on the other hand, is insensitive to the start value in *linear problems*.)

7.11 Example: Adaptive Runge-Kutta methods

Odespy also offers solution methods that can adapt the size of Δt with time to match a desired accuracy in the solution. Intuitively, small time steps will be chosen in areas where the solution is changing rapidly, while larger time steps can be used where the solution is slowly varying. Some kind of *error estimator* is used to adjust the next time step at each time level.

A very popular adaptive method for solving ODEs is the Dormand-Prince Runge-Kutta method of order 4 and 5. The 5th-order method is used as a reference solution and the difference between the 4th- and 5th-order methods is used as an indicator of the error in the numerical solution. The Dormand-Prince method is the default choice in MATLAB's widely used `ode45` routine.

We can easily set up Odespy to use the Dormand-Prince method and see how it selects the optimal time steps. To this end, we request only one time step from $t = 0$ to $t = T$ and ask the method to compute the necessary non-uniform time mesh to meet a certain error tolerance. The code goes like

```
import odespy
import numpy as np
import decay_mod
import sys
#import matplotlib.pyplot as plt
import scitools.std as plt

def f(u, t):
    return -a*u

def exact_solution(t):
    return I*np.exp(-a*t)

I = 1; a = 2; T = 5
tol = float(sys.argv[1])
solver = odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

Nt = 1  # just one step - let the scheme find its intermediate points
t_mesh = np.linspace(0, T, Nt+1)
t_fine = np.linspace(0, T, 10001)

solver.set_initial_condition(I)
u, t = solver.solve(t_mesh)

# u and t will only consist of [I, u^Nt] and [0,T]
# solver.u_all and solver.t_all contains all computed points
plt.plot(solver.t_all, solver.u_all, 'ko')
plt.hold('on')
plt.plot(t_fine, exact_solution(t_fine), 'b-')
plt.legend(['tol=%.0E' % tol, 'exact'])
plt.savefig('tmp_odespy_adaptive.png')
plt.show()
```

Running four cases with tolerances 10^{-1}, 10^{-3}, 10^{-5}, and 10^{-7}, gives the results in Figure 18. Intuitively, one would expect denser points in the beginning of the decay and larger time steps when the solution flattens out.

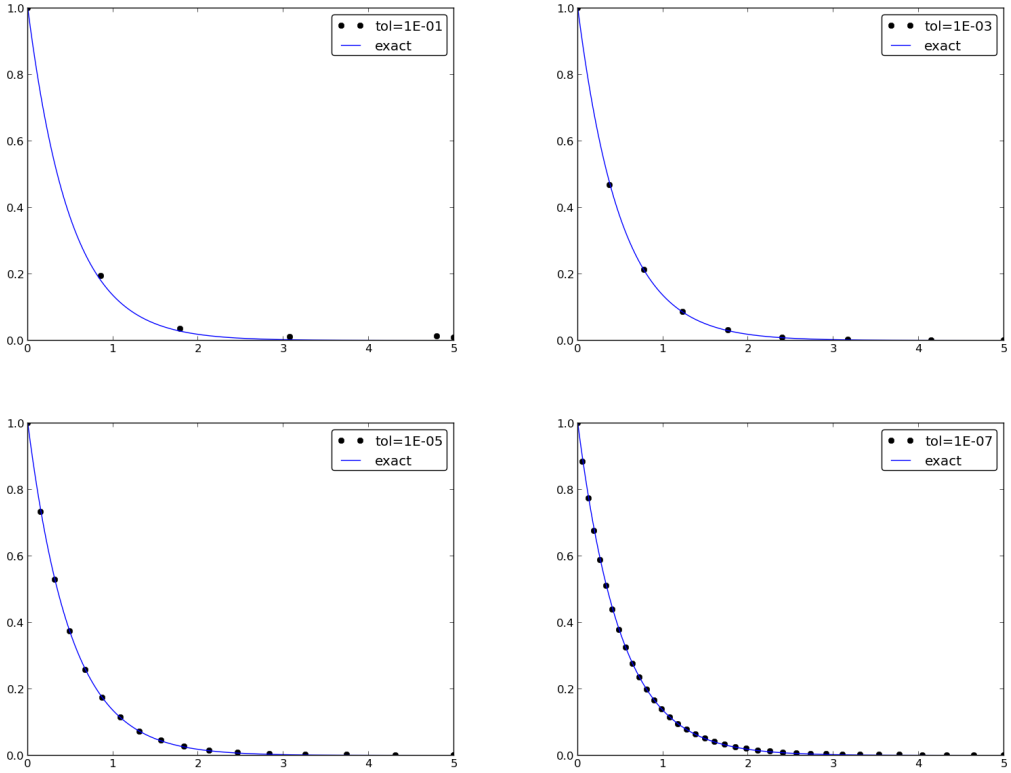


Figure 18: Choice of adaptive time mesh by the Dormand-Prince method for different tolerances.

8 Exercises

Exercise 9: Experiment with precision in tests and the size of u

It is claimed in Section 6.5 that most numerical methods will reproduce a linear exact solution to machine precision. Test this assertion using the test function `test_linear_solution` in the `decay_vc.py` program. Vary the parameter `c` from very small, via `c=1` to many larger values, and print out the maximum difference between the numerical solution and the exact solution. What is the relevant value of the tolerance in the float comparison in each case? Filename: `test_precision`.

Exercise 10: Implement the 2-step backward scheme

Implement the 2-step backward method (113) for the model $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. Allow the first step to be computed by either the Backward Euler scheme or the Crank-Nicolson scheme. Verify the implementation by choosing $a(t)$ and $b(t)$ such that the exact solution is linear in t (see Section 6.5). Show mathematically that a linear solution is indeed a solution of the discrete equations.

Compute convergence rates (see Section 6.6) in a test case using $a = \text{const}$ and $b = 0$, where we easily have an exact solution, and determine if the choice of a first-order scheme (Backward Euler) for the first step has any impact on the overall accuracy of this scheme. The expected error goes like $\mathcal{O}(\Delta t^2)$. Filename: `decay_backward2step`.

Exercise 11: Implement the 2nd-order Adams-Bashforth scheme

Implement the 2nd-order Adams-Bashforth method (120) for the decay problem $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Use the Forward Euler method for the first step such that the overall scheme is explicit. Verify the implementation using an exact solution that is linear in time. Analyze the scheme by searching for solutions $u^n = A^n$ when $a = \text{const}$ and $b = 0$. Compare this second-order secheme to the Crank-Nicolson scheme. Filename: `decay_AdamsBashforth2`.

Exercise 12: Implement the 3rd-order Adams-Bashforth scheme

Implement the 3rd-order Adams-Bashforth method (121) for the decay problem $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Since the scheme is explicit, allow it to be started by two steps with the Forward Euler method. Investigate experimentally the case where $b = 0$ and a is a constant: Can we have oscillatory solutions for large Δt? Filename: `decay_AdamsBashforth3`.

Exercise 13: Analyze explicit 2nd-order methods

Show that the schemes (118) and (119) are identical in the case $f(u, t) = -a$, where $a > 0$ is a constant. Assume that the numerical solution reads $u^n = A^n$ for some unknown amplification factor A to be determined. Find A and derive stability criteria. Can the scheme produce oscillatory solutions of $u' = -au$? Plot the numerical and exact amplification factor. Filename: `decay_RK2_Taylor2`.

Problem 14: Implement and investigate the Leapfrog scheme

A Leapfrog scheme for the ODE $u'(t) = -a(t)u(t) + b(t)$ is defined by

$$[D_{2t}u = -au + b]^n .$$

A separate method is needed to compute u^1. The Forward Euler scheme is a possible candidate.

a) Implement the Leapfrog scheme for the model equation. Plot the solution in the case $a = 1$, $b = 0$, $I = 1$, $\Delta t = 0.01$, $t \in [0, 4]$. Compare with the exact solution $u_{\mathrm{e}}(t) = e^{-t}$.

b) Show mathematically that a linear solution in t fulfills the Forward Euler scheme for the first step and the Leapfrog scheme for the subsequent steps. Use this linear solution to verify the implementation, and automate the verification through a test function.

Hint. It can be wise to automate the calculations such that it is easy to redo the calculations for other types of solutions. Here is a possible `sympy` function that takes a symbolic expression `u` (implemented as a Python function of `t`), fits the `b` term, and checks if `u` fulfills the discrete equations:

```
import sympy as sym

def analyze(u):
    t, dt, a = sym.symbols('t dt a')

    print 'Analyzing u_e(t)=%s' % u(t)
    print 'u(0)=%s' % u(t).subs(t, 0)

    # Fit source term to the given u(t)
    b = sym.diff(u(t), t) + a*u(t)
    b = sym.simplify(b)
    print 'Source term b:', b

    # Residual in discrete equations; Forward Euler step
    R_step1 = (u(t+dt) - u(t))/dt + a*u(t) - b
    R_step1 = sym.simplify(R_step1)
    print 'Residual Forward Euler step:', R_step1

    # Residual in discrete equations; Leapfrog steps
    R = (u(t+dt) - u(t-dt))/(2*dt) + a*u(t) - b
    R = sym.simplify(R)
    print 'Residual Leapfrog steps:', R

def u_e(t):
    return c*t + I

analyze(u_e)
# or short form: analyze(lambda t: c*t + I)
```

c) Show that a second-order polynomial in t cannot be a solution of the discrete equations. However, if a Crank-Nicolson scheme is used for the first step, a second-order polynomial solves the equations exactly.

d) Create a manufactured solution $u(t) = \sin(t)$ for the ODE $u' = -au + b$. Compute the convergence rate of the Leapfrog scheme using this manufactured solution. The expected convergence rate of the Leapfrog scheme is $\mathcal{O}(\Delta t^2)$. Does the use of a 1st-order method for the first step impact the convergence rate?

e) Set up a set of experiments to demonstrate that the Leapfrog scheme (115) is associated with numerical artifacts (instabilities). Document the main results from this investigation.

f) Analyze and explain the instabilities of the Leapfrog scheme (115):

1. Choose $a = \text{const}$ and $b = 0$. Assume that an exact solution of the discrete equations has the form $u^n = A^n$, where A is an amplification factor to be determined. Derive an equation for A by inserting $u^n = A^n$ in the Leapfrog scheme.
2. Compute A either by hand and/or with the aid of `sympy`. The polynomial for A has two roots, A_1 and A_2. Let u^n be a linear combination $u^n = C_1 A_1^n + C_2 A_2^n$.
3. Show that one of the roots is the reason for instability.
4. Compare A with the exact expression, using a Taylor series approximation.
5. How can C_1 and C_2 be determined?

g) Since the original Leapfrog scheme is unconditionally unstable as time grows, it demands some stabilization. This can be done by filtering, where we first find u^{n+1} from the original Leapfrog scheme and then replace u^n by $u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})$, where γ can be taken as 0.6. Implement the filtered Leapfrog scheme and check that it can handle tests where the original Leapfrog scheme is unstable.
Filename: `decay_leapfrog`.

Problem 15: Make a unified implementation of many schemes

Consider the linear ODE problem $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. Explicit schemes for this problem can be written in the general form

$$u^{n+1} = \sum_{j=0}^{m} c_j u^{n-j}, \tag{126}$$

for some choice of $c_0, \ldots, c_m$. Find expressions for the c_j coefficients in case of the θ-rule, the three-level backward scheme, the Leapfrog scheme, the 2nd-order Runge-Kutta method, and the 3rd-order Adams-Bashforth scheme.

Make a class `ExpDecay` that implements the general updating formula (126). The formula cannot be applied for $n < m$, and for those n values, other schemes must be used. Assume for simplicity that we just repeat Crank-Nicolson steps until (126) can be used. Use a subclass to specify the list $c_0, \ldots, c_m$ for a particular method, and implement subclasses for all the mentioned schemes. Verify the implementation by testing with a linear solution, which should be exactly reproduced by all methods. Filename: `decay_schemes_unified`.

9 Applications of exponential decay models

This section presents many mathematical models that all end up with ODEs of the type $u' = -au + b$. The applications are taken from biology, finance, and physics, and cover population growth or decay, compound interest and inflation, radioactive decay, cooling of objects, compaction of geological media, pressure variations in the atmosphere, and air resistance on falling or rising bodies.

Before we turn to the applications, however, we take a brief look at the technique of scaling, which is so useful in many applications.

9.1 Scaling

Real applications of a model $u' = -au + b$ will often involve a lot of parameters in the expressions for a and b. It can be quite a challenge to find relevant values of all parameters. In simple problems, however, it turns out that it is not always necessary to estimate all parameters because we can lump them into one or a few *dimensionless* numbers by using a very attractive technique called scaling. It simply means to stretch the u and t axis in the present problem - and suddenly all parameters in the problem are lumped into one parameter if $b \neq 0$ and no parameter when $b = 0$!

Scaling means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

where u_m is a characteristic value of u, u_c is a characteristic size of the range of u values, and t_c is a characteristic size of the range of t where u shows significant variation. Choosing u_m, u_c, and t_c is not always easy and is often an art in complicated problems. We just state one choice first:

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

Inserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a and b are constants, results (after some algebra) in the *scaled problem*

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u}, \quad \bar{u}(0) = 1 - \beta,$$

where β is a dimensionless number

$$\beta = \frac{b}{Ia}\,.$$

That is, only the special combination of $b/(Ia)$ matters, not what the individual values of b, a, and I are. Moreover, if $b = 0$, the scaled problem is independent of a and I! In practice this means that we can perform one numerical simulation of the scaled problem and recover the solution of any problem for a given a and I by stretching the axis in the plot: $u = I\bar{u}$ and $t = \bar{t}/a$. For $b \neq 0$, we simulate the scaled problem for a few β values and recover the physical solution u by translating and stretching the u axis and stretching the t axis.

The scaling breaks down if $I = 0$. In that case we may choose $u_m = 0$, $u_c = b/a$, and $t_c = 1/b$, resulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

As with $b = 0$, the case $I = 0$ has a scaled problem with no physical parameters!

It is common to drop the bars after scaling and write the scaled problem as $u' = -u$, $u(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementation of the problem $u' = -au + b$, $u(0) = I$, can be reused for the scaled problem by setting $a = 1$, $b = 0$, and $I = 1 - \beta$ in the code, if $I \neq 0$, or one sets $a = 1$, $b = 1$, and $I = 0$ when the physical I is zero. Falling bodies in fluids, as described in Section 9.10, involves $u' = -au + b$ with seven physical parameters. All these vanish in the scaled version of the problem if we start the motion from rest!

Many more details about scaling is presented in [5].

9.2 Evolution of a population

Let N be the number of individuals in a population occupying some spatial domain. Despite N being an integer in this problem, we shall compute with N as a real number and view $N(t)$ as a continuous function of time. The basic model assumption is that in a time interval Δt the number of newcomers to the populations (newborns) is proportional to N, with proportionality constant $\bar{b}$. The amount of newcomers will increase the population and result in

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

It is obvious that a long time interval Δt will result in more newcomers and hence a larger $\bar{b}$. Therefore, we introduce $b = \bar{b}/\Delta t$: the number of newcomers per unit time and per individual. We must then multiply b by the length of the time interval considered and by the population size to get the total number of new individuals, $b\Delta tN$.

If the number of removals from the population (deaths) is also proportional to N, with proportionality constant $d\Delta t$, the population evolves according to

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t)\,.$$

Dividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b - d)N, \quad N(0) = N_0\,. \tag{127}$$

In a population where the death rate (d) is larger than then newborn rate (b), $b - d < 0$, and the population experiences exponential decay rather than exponential growth.

In some populations there is an immigration of individuals into the spatial domain. With I individuals coming in per time unit, the equation for the population change becomes

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t) + \Delta tI\,.$$

The corresponding ODE reads

$$N' = (b-d)N + I, \quad N(0) = N_0\,. \tag{128}$$

Emigration is also modeled by this I term if we just change its sign: $I < 0$. So, the I term models migration in and out of the domain in general.

Some simplification arises if we introduce a fractional measure of the population: $u = N/N_0$ and set $r = b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1, \tag{129}$$

where $f = I/N_0$ measures the net immigration per time unit as the fraction of the initial population. Very often, r is approximately constant, but f is usually a function of time.

The growth rate r of a population decreases if the environment has limited resources. Suppose the environment can sustain at most $N_{\max}$ individuals. We may then assume that the growth rate approaches zero as N approaches $N_{\max}$, i.e., as u approaches $M = N_{\max}/N_0$. The simplest possible evolution of r is then a linear function: $r(t) = \varrho(1 - u(t)/M)$, where ϱ is the initial growth rate when the population is small relative to the maximum size and there is enough resources. Using this $r(t)$ in (129) results in the *logistic model* for the evolution of a population (assuming for the moment that $f = 0$):

$$u' = \varrho(1 - u/M)u, \quad u(0) = 1\,. \tag{130}$$

Initially, u will grow at rate ϱ, but the growth will decay as u approaches M, and then there is no more change in u, causing $u \to M$ as $t \to \infty$. Note that the logistic equation $u' = \varrho(1 - u/M)u$ is *nonlinear* because of the quadratic term $-u^2\varrho/M$.

9.3 Compound interest and inflation

Say the annual interest rate is r percent and that the bank adds the interest once a year to your investment. If u^n is the investment in year n, the investment in year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

In reality, the interest rate is added every day. We therefore introduce a parameter m for the number of periods per year when the interest is added. If n counts the periods, we have the fundamental model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,. \tag{131}$$

This model is a *difference equation*, but it can be transformed to a continuous differential equation through a limit process. The first step is to derive a formula

for the growth of the investment over a time t. Starting with an investment u^0, and assuming that r is constant in time, we get

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right) u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

Introducing time t, which here is a real-numbered counter for years, we have that $n = mt$, so we can write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0 \,.$$

The second step is to assume *continuous compounding*, meaning that the interest is added continuously. This implies $m \to \infty$, and in the limit one gets the formula

$$u(t) = u_0 e^{rt/100}, \tag{132}$$

which is nothing but the solution of the ODE problem

$$u' = \frac{r}{100} u, \quad u(0) = u_0 \,. \tag{133}$$

This is then taken as the ODE model for compound interest if $r > 0$. However, the reasoning applies equally well to inflation, which is just the case $r < 0$. One may also take the r in (133) as the net growth of an investemt, where r takes both compound interest and inflation into account. Note that for real applications we must use a time-dependent r in (133).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is then a process exhibiting exponential decay according to

$$u' = -au, \quad u(0) = I \,.$$

9.4 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, experience shows that the temperature falls rapidly in the beginning, and then the change in temperature levels off until the body's temperature equals that of the surroundings. Newton carried out some experiments on cooling hot iron and found that the temperature evolved as a "geometric progression at times in arithmetic progression", meaning that the temperature decayed exponentially. Later, this result was formulated as a differential equation: the rate of change of the temperature in a body is proportional to the temperature difference between the body and its surroundings. This statement is known as *Newton's law of cooling*, which mathematically can be expressed as

$$\frac{dT}{dt} = -k(T - T_s), \tag{134}$$

where T is the temperature of the body, T_s is the temperature of the surroundings (which may be time-dependent), t is time, and k is a positive constant. Equation (134) is primarily viewed as an empirical law, valid when heat is efficiently convected away from the surface of the body by a flowing fluid such as air at constant temperature T_s. The *heat transfer coefficient* k reflects the transfer of heat from the body to the surroundings and must be determined from physical experiments.

The cooling law (134) needs an initial condition $T(0) = T_0$.

9.5 Radioactive decay

An atomic nucleus of an unstable atom may lose energy by emitting ionizing particles and thereby be transformed to a nucleus with a different number of protons and neutrons. This process is known as radioactive decay. Actually, the process is stochastic when viewed for a single atom, because it is impossible to predict exactly when a particular atom emits a particle. Nevertheless, with a large number of atoms, N, one may view the process as deterministic and compute the mean behavior of the decay. Below we reason intuitively about an ODE for the mean behavior. Thereafter, we show mathematically that a detailed stochastic model for single atoms leads to the same mean behavior.

Deterministic model. Suppose at time t, the number of the original atom type is $N(t)$. A basic model assumption is that the transformation of the atoms of the original type in a small time interval Δt is proportional to N, so that

$$N(t + \Delta t) = N(t) - a\Delta t N(t),$$

where $a > 0$ is a constant. Introducing $u = N(t)/N(0)$, dividing by Δt and letting $\Delta t \to 0$ gives the following ODE:

$$u' = -au, \quad u(0) = 1\,. \tag{135}$$

The parameter a can for a given nucleus be expressed through the *half-life* $t_{1/2}$, which is the time taken for the decay to reduce the initial amount by one half, i.e., $u(t_{1/2}) = 0.5$. With $u(t) = e^{-at}$, we get $t_{1/2} = a^{-1}\ln 2$ or $a = \ln 2/t_{1/2}$.

Stochastic model. Originally, we have N_0 atoms. Up to some particular time t, each atom may either have decayed or not. If not, they have "survived". We want to count how many original atoms that have survived. The survival of a single atom at time t is a random event. Since there are only two outcomes, survival or decay, we have a Bernoulli trial. Let p be the probability of survival (implying that the probability of decay is $1 - p$). If each atom survives independently of the others, and the probability of survival is the same for every atom, we have N_0 Bernoulli trials, known as a *binomial experiment* from probability

theory. The probability $P(N)$ that N out of the N_0 atoms have survived at time t is then given by the famous *binomial distribution*

$$P(N) = \frac{N_0!}{N!(N_0 - N)!} p^N (1-p)^{N_0 - N}.$$

The mean (or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be $N_0 p$.

It remains to estimate p. Let the interval $[0, t]$ be divided into m small subintervals of length Δt. We make the assumption that the probability of decay of a single atom in an interval of length Δt is $\tilde{p}$, and that this probability is proportional to Δt: $\tilde{p} = \lambda \Delta t$ (it sounds natural that the probability of decay increases with Δt). The corresponding probability of survival is $1 - \lambda \Delta t$. Believing that λ is independent of time, we have, for each interval of length Δt, a Bernoulli trial: the atom either survives or decays in that interval. Now, p should be the probability that the atom survives in all the intervals, i.e., that we have m successful Bernoulli trials in a row and therefore

$$p = (1 - \lambda \Delta t)^m.$$

The expected number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0 p = N_0 (1 - \lambda \Delta t)^m, \quad m = t/\Delta t. \tag{136}$$

To see the relation between the two types of Bernoulli trials and the ODE above, we go to the limit $\Delta t \to t$, $m \to \infty$. One can show that

$$p = \lim_{m \to \infty} (1 - \lambda \Delta t)^m = \lim_{m \to \infty} \left(1 - \lambda \frac{t}{m}\right)^m = e^{-\lambda t}$$

This is the famous exponential waiting time (or arrival time) distribution for a Poisson process in probability theory (obtained here, as often done, as the limit of a binomial experiment). The probability of decay, $1 - e^{-\lambda t}$, follows an exponential distribution. The limit means that m is very large, hence Δt is very small, and $\tilde{p} = \lambda \Delta t$ is very small since the intensity of the events, λ, is assumed finite. This situation corresponds to a very small probability that an atom will decay in a very short time interval, which is a reasonable model. The same model occurs in lots of different applications, e.g., when waiting for a taxi, or when finding defects along a rope.

Relation between stochastic and deterministic models. With $p = e^{-\lambda t}$ we get the expected number of original atoms at t as $N_0 p = N_0 e^{-\lambda t}$, which is exactly the solution of the ODE model $N' = -\lambda N$. This also gives an interpretation of a via λ or vice versa. Our important finding here is that the ODE model captures the mean behavior of the underlying stochastic model. This is, however, not always the common relation between microscopic stochastic models and macroscopic "averaged" models.

Also of interest, is that a Forward Euler discretization of $N' = -\lambda N$, $N(0) = N_0$, gives $N^m = N_0(1 - \lambda \Delta t)^m$ at time $t_m = m \Delta t$, which is exactly the expected

value of the stochastic experiment with N_0 atoms and m small intervals of length Δt, where each atom can decay with probability $\lambda\Delta t$ in an interval.

A fundamental question is how accurate the ODE model is. The underlying stochastic model fluctuates around its expected value. A measure of the fluctuations is the standard deviation of the binomial experiment with N_0 atoms, which can be shown to be $\mathrm{Std}[P] = \sqrt{N_0 p(1-p)}$. Compared to the size of the expectation, we get the normalized standard deviation

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1}-1} = N_0^{-1/2}\sqrt{(1-e^{-\lambda t})^{-1}-1} \approx (N_0\lambda t)^{-1/2},$$

showing that the normalized fluctuations are very small if N_0 is very large, which is usually the case.

9.6 Chemical kinetics

Irreversible reaction of two substances. Consider two chemical substances, A and B, and a chemical reaction that turns A into B. In a small time interval, some of the molecules of type A are transformed into molecules of B. This process is, from a mathematical modeling point of view, equivalent to the radioactive decay process described in the previous section. We can therefore apply the same modeling approach. If N_A is the number of molecules of substance A, we have that N_A is governed by the differential equation

$$\frac{dN_A}{dt} = -kN_A,$$

where (the constant) k is called the *rate constant* of the reaction. Rather than using the number of molecules, we use the *concentration* of molecules: $[A](t) = N_A(t)/N_A(0)$. We see that $d[A]/dt = N_A(0)^{-1}dN_A/dt$. Replacing N_A by $[A]$ in the equation for N_A leads to the equation for the concentration $[A]$:

$$\frac{d[A]}{dt} = -k[A], \quad t \in (0,T],\ [A](0) = A_0,\ . \tag{137}$$

Since substance A is transformed to substance B, we have that the concentration of $[B]$ grows by the loss of $[A]$:

$$\frac{d[B]}{dt} = k[A], \quad [B](0) = B_0\,.$$

The mathematical model can either be (137) or the system

$$\frac{d[A]}{dt} = -k[A], \qquad t \in (0,T] \tag{138}$$

$$\frac{d[B]}{dt} = k[A], \qquad t \in (0,T] \tag{139}$$

$$[A](0) = A_0, \tag{140}$$

$$[B](0) = B_0\,. \tag{141}$$

This reaction is known as a *first-order reaction*, where each molecule of A makes an independent decision about whether to complete the reaction, i.e., independent of what happens to any other molecule.

An n-th order reaction is modeled by

$$\frac{d[A]}{dt} = -k[A]^n, \tag{142}$$

$$\frac{d[B]}{dt} = k[A]^n, \tag{143}$$

for $t \in (0, T]$ with initial conditions $[A](0) = A_0$ and $[B](0) = B_0$. Here, n can be a real number, but is most often an integer. Note that the sum of the concentrations is constant since

$$\frac{d[A]}{dt} + \frac{d[B]}{dt} = 0 \quad \Rightarrow \quad [A](t) + [B](t) = \text{const} \quad \Rightarrow \quad [A](t) + [B](t) = [A](0) + [B](0) = A_0 + B_0 \,.$$

Reversible reaction of two substances. Let the chemical reaction turn substance A into B and substance B into A. The rate of change of $[A]$ has then two contributions: a loss $k_A[A]$ and a gain $k_B[B]$:

$$\frac{d[A]}{dt} = -k_A[A] + k_B[B], \quad t \in (0, T], \ [A](0) = A_0 \,. \tag{144}$$

Similarly for substance B,

$$\frac{d[A]}{dt} = k_A[A] - k_B[B], \quad t \in (0, T], \ [A](0) = A_0 \,. \tag{145}$$

Again,

$$\frac{d[A]}{dt} + \frac{d[B]}{dt} = 0 \quad \Rightarrow \quad [A](t) + [B](t) = A_0 + B_0 \,.$$

Irreversible reaction of two substances into a third. Now we consider two chemical substances, A and B, reacting with each other and producing a substance C. In a small time interval Δt, molecules of type A and B are occasionally colliding, and in some of the collisions, a chemical reaction occurs, which turns A and B into a molecule of type C. (More generally, M_A molecules of A and M_B molecules of B react to form M_C molecules of C.) The number of possible pairings, and thereby collisions, of A and B is $N_A N_B$, where N_A is the number of molecules of A, and N_B is the number of molecules of N_B. A fraction k of these collisions, $\hat{k}\Delta t N_A N_B$, features a chemical reaction and produce N_C molecules of C. The fraction is thought to be proportional to Δt: considering a twice as long time interval, twice as many molecules collide, and twice as many reactions occur. The increase in molecules of substance C is now found from the reasoning

$$N_C(t+\Delta t) = N_C(t) + \hat{k}\Delta t N_A N_B\,.$$

Dividing by Δt,

$$\frac{N_C(t+\Delta t) - N_C(t)}{\Delta t} = \hat{k} N_A N_B,$$

and letting $\Delta t \to 0$, gives the differential equation

$$\frac{dN_C}{dt} = \hat{k} N_A N_B\,.$$

(This equation is known as the important "law of mass action" discovered by the Norwegian scientists Cato M. Guldberg and Peter Waage. A more general form of the right-hand side is $\hat{k} N_A^\alpha N_B^\beta$. All the constants $\hat{k}$, α, and β must be determined from experiments.)

Working with concentrations instead, we introduce $[C](t) = N_C(t)/N_C(0)$, with similar definitions for $[A]$ and $[B]$ we get

$$\frac{d[C]}{dt} = k[A][B]\,. \tag{146}$$

The constant k is related to $\hat{k}$ by $k = \hat{k} N_A(0) N_B(0)/N_C(0)$. The gain in C is a loss of A and B:

$$\frac{d[A]}{dt} = -k[A][B], \tag{147}$$

$$\frac{d[B]}{dt} = -k[A][B]\,. \tag{148}$$

A biochemical reaction. A common reaction (known as Michaelis-Menton kinetics) turns a substrate S into a product P with aid of an enzyme E. The reaction is a two-stage process: first S and E reacts to form a complex ES, where the enzyme and substrate are bound to each other, and then ES is turned into E and P. In the first stage, S and E react to produce a growth of ES according to the law of mass action:

$$\frac{d[S]}{dt} = -k_+[E][S],$$

$$\frac{d[P]}{dt} = -k_+[E][S],$$

$$\frac{d[ES]}{dt} = k_+[E][S]\,.$$

The complex ES reacts and produces the product P at rate $-k_v[ES]$ and E at rate $-k_-[ES]$. The total set of reactions can then be expressed by

$$\frac{d[ES]}{dt} = k_+[E][S] - k_v[ES] - k_-[ES], \tag{149}$$
$$\frac{d[P]}{dt} = k_v[ES], \tag{150}$$
$$\frac{d[S]}{dt} = -k_+[E][S] + k_-[ES], \tag{151}$$
$$\frac{d[E]}{dt} = -k_+[E][S] + k_-[ES] + k_v[ES]\,. \tag{152}$$

The initial conditions are $[ES](0) = [P](0) = 0$, and $[S] = S_0$, $[E] = E_0$. The constants k_+, k_-, and k_v must be determined from experiments.

9.7 Spreading of diseases

The modeling of spreading of diseases is very similar to the modeling of chemical reactions in Section 9.6. The field of epidemiology speaks about susceptibles: people who can get a disease; infectives: people who are infected and can infect susceptibles; and recovered: people who have recovered from the disease and become immune. Three categories are accordingly defined: S for susceptibles, I for infectives, and R for recovered. The number in each category is tracked by the functions $S(t)$, $I(t)$, and $R(t)$.

To model how many people that get infected in a small time interval Δt, we reason as with reactions in Section 9.6. The possible number of parings ("collisions") between susceptibles and infected is SI. A fraction of these, $\beta\Delta tSI$, will actually meet and the infected succeeds of infecting the susceptible, where β is a parameter to be empirically estimated. This leads to a loss of susceptibles and a gain of infected:

$$S(t+\Delta t) = S(t) - \beta\Delta tSI,$$
$$I(t+\Delta t) = I(t) + \beta\Delta tSI\,.$$

In the same time interval, a fraction $\nu\Delta tI$ of the infected is recovered. The parameter ν^{-1} has the interpretation of the average length of the disease (time to recovery). The $\nu\Delta tI$ term is a loss for the I category, but a gain for the R category:

$$I(t+\Delta t) = I(t) + \beta\Delta tSI - \nu\Delta tI, R(t+\Delta t) \qquad = R(t) + \nu\Delta tI\,.$$

Dividing these equations by Δt and going to the limit $\Delta t \to 0$, gives the ODE system

$$\frac{dS}{dt} = -\beta SI, \tag{153}$$

$$\frac{dI}{dt} = \beta SI - \nu I, \tag{154}$$

$$\frac{dR}{dt} = \nu I, \tag{155}$$

with initial values $S(0) = S_0$, $I(0) = I_0$, and $R(0) = 0$. By adding the equations, we realize that

$$\frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0 \quad \Rightarrow \quad S + I + R = \text{const} = N,$$

where N is the total number in the population under consideration. This property can be used as a partial verification during simulations.

Equations (153)-(155) are known as the SIR model in epidemiology. The model can easily be extended to incorporate vaccination programs, loss immunity after some time, etc. Typical diseases that can be simulated by the SIR model and its variants are measles, smallpox, flu, plague, and HIV.

9.8 Decay of atmospheric pressure with altitude

Vertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\varrho g\,. \tag{156}$$

Here, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807$ m/s^2 is a standard value of the acceleration of gravity. (Equation (156) follows directly from the general Navier-Stokes equations for fluid motion, with the assumption that the air does not move.)

The pressure is related to density and temperature through the ideal gas law

$$\varrho = \frac{Mp}{R^*T}, \tag{157}$$

where M is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal gas constant (8.314 Nm/(mol K)), and T is the temperature in Kelvin. All variables p, ϱ, and T vary with the height z. Inserting (157) in (156) results in an ODE with a variable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,. \tag{158}$$

Multiple atmospheric layers. The atmosphere can be approximately modeled by seven layers. In each layer, (158) is applied with a linear temperature of the form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

where $z = h_i$ denotes the bottom of layer number i, having temperature $\bar{T}_i$, and L_i is a constant in layer number i. The table below lists h_i (m), $\bar{T}_i$ (K), and L_i (K/m) for the layers $i = 0, \ldots, 6$.

i	h_i	$\bar{T}_i$	L_i
0	0	288	-0.0065
1	11,000	216	0.0
2	20,000	216	0.001
3	32,000	228	0.0028
4	47,000	270	0.0
5	51,000	270	-0.0028
6	71,000	214	-0.002

For implementation it might be convenient to write (158) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p, \tag{159}$$

where $\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values given in the table. The value of the pressure at the sea level $z = 0$, $p_0 = p(0)$, is 101325 Pa.

Simplification: $L = 0$. One common simplification is to assume that the temperature is constant within each layer. This means that $L = 0$.

Simplification: one-layer model. Another commonly used approximation is to work with one layer instead of seven. This one-layer model is based on $T(z) = T_0 - Lz$, with sea level standard temperature $T_0 = 288$ K and temperature lapse rate $L = 0.0065$ K/m.

9.9 Compaction of sediments

Sediments, originally made from materials like sand and mud, get compacted through geological time by the weight of new material that is deposited on the sea bottom. The porosity ϕ of the sediments tells how much void (fluid) space there is between the sand and mud grains. The porosity drops with depth, due to the weight of the sediments above. This makes the void space shrink, and thereby compaction increases.

A typical assumption is that the change in ϕ at some depth z is negatively proportional to ϕ. This assumption leads to the differential equation problem

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0, \tag{160}$$

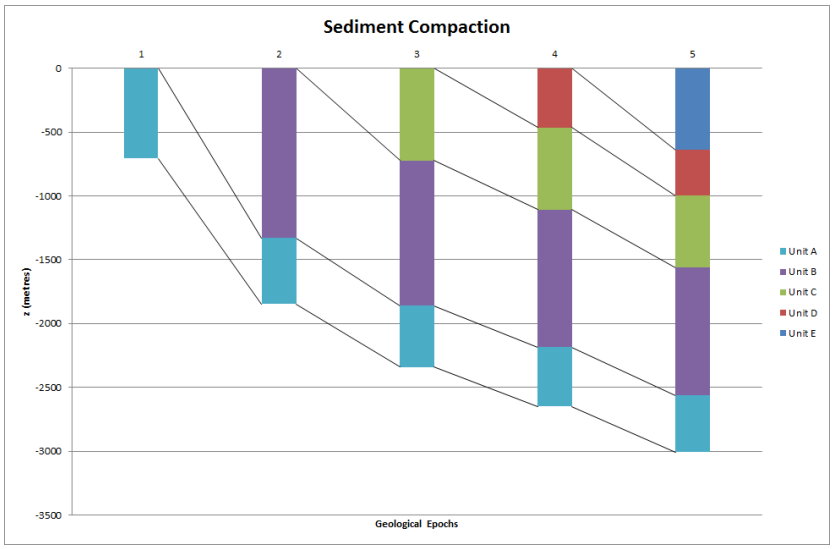


Figure 19: Illustration of the compaction of geological layers (with different colors) through time.

where the z axis points downwards, $z = 0$ is the surface with known porosity, and $c > 0$ is a constant.

The upper part of the Earth's crust consists of many geological layers stacked on top of each other, as indicated in Figure 19. The model (160) can be applied for each layer. In layer number i, we have the unknown porosity function $\phi_i(z)$ fulfilling $\phi_i'(z) = -c_i z$, since the constant c in the model (160) depends on the type of sediment in the layer. Alternatively, we can use (160) to describe the porosity through all layers if c is taken as a piecewise constant function of z, equal to c_i in layer i. From the figure we see that new layers of sediments are deposited on top of older ones as time progresses. The compaction, as measured by ϕ, is rapid in the beginning and then decreases (exponentially) with depth in each layer.

When we drill a well at present time through the right-most column of sediments in Figure 19, we can measure the thickness of the sediment in (say) the bottom layer. Let L_1 be this thickness. Assuming that the volume of sediment remains constant through time, we have that the initial volume, $\int_0^{L_{1,0}} \phi_1 dz$, must equal the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is the depth of the bottom of the sediment in the present day configuration. After having solved for ϕ_1 as a function of z, we can then find the original thickness $L_{1,0}$ of the sediment from the equation

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

In hydrocarbon exploration it is important to know $L_{1,0}$ and the compaction history of the various layers of sediments.

9.10 Vertical motion of a body in a viscous fluid

A body moving vertically through a fluid (liquid or gas) is subject to three different types of forces: the gravity force, the drag force, and the buoyancy force.

Overview of forces. Taking the upward direction as positive, the gravity force is $F_g = -mg$, where m is the mass of the body and g is the acceleration of gravity. The uplift or buoyancy force ("Archimedes force") is $F_b = \varrho g V$, where ϱ is the density of the fluid and V is the volume of the body.

The drag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu}, \tag{161}$$

where d is the diameter of the body in the direction perpendicular to the flow, v is the velocity of the body, and μ is the dynamic viscosity of the fluid. When $\mathrm{Re} < 1$, the drag force is fairly well modeled by the so-called Stokes' drag, which for a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,. \tag{162}$$

Quantities are taken as positive in the upwards vertical direction, so if $v > 0$ and the body moves upwards, the drag force acts downwards and become negative, in accordance with the minus sign in expression for $F_d^{(S)}$.

For large Re, typically $\mathrm{Re} > 10^3$, the drag force is quadratic in the velocity:

$$F_d^{(q)} = -\frac{1}{2} C_D \varrho A|v|v, \tag{163}$$

where C_D is a dimensionless drag coefficient depending on the body's shape, and A is the cross-sectional area as produced by a cut plane, perpendicular to the motion, through the thickest part of the body. The superscripts ^{q} and ^{S} in $F_d^{(S)}$ and $F_d^{(q)}$ indicate Stokes drag and quadratic drag, respectively.

Equation of motion. All the mentioned forces act in the vertical direction. Newton's second law of motion applied to the body says that the sum of these forces must equal the mass of the body times its acceleration a in the vertical direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

Here we have chosen to model the fluid resistance by the Stokes drag. Inserting the expressions for the forces yields

$$ma = -mg - 3\pi d\mu v + \varrho g V\,.$$

The unknowns here are v and a, i.e., we have two unknowns but only one equation. From kinematics in physics we know that the acceleration is the time derivative of the velocity: $a = dv/dt$. This is our second equation. We can easily eliminate a and get a single differential equation for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho g V\,.$$

A small rewrite of this equation is handy: We express m as $\varrho_b V$, where ϱ_b is the density of the body, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V}v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{164}$$

We may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right), \tag{165}$$

so that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,. \tag{166}$$

The corresponding initial condition is $v(0) = v_0$ for some prescribed starting velocity v_0.

This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, leading to the result

$$v'(t) = -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{167}$$

Defining

$$a = \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}, \tag{168}$$

and b as above, we can write (167) as

$$v'(t) = -a|v|v + b\,. \tag{169}$$

Terminal velocity. An interesting aspect of (166) and (169) is whether v will approach a final constant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A constant v means that $v'(t) \to 0$ as $t \to \infty$ and therefore the terminal velocity v_T solves

$$0 = -av_T + b$$

and

$$0 = -a|v_T|v_T + b\,.$$

The former equation implies $v_T = b/a$, while the latter has solutions $v_T = -\sqrt{|b|/a}$ for a falling body ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

A Crank-Nicolson scheme. Both governing equations, the Stokes' drag model (166) and the quadratic drag model (169), can be readily solved by the Forward Euler scheme. For higher accuracy one can use the Crank-Nicolson method, but a straightforward application of this method gives a nonlinear equation in the new unknown value v^{n+1} when applied to (169):

$$\frac{v^{n+1} - v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,. \tag{170}$$

The first term on the right-hand side of (170) is the arithmetic average of $-|v|v$ evaluated at time levels n and $n+1$.

Instead of approximating the term $-|v|v$ by an arithmetic average, we can use a *geometric mean*:

$$(|v|v)^{n+\frac{1}{2}} \approx |v^n|v^{n+1}\,. \tag{171}$$

The error is of second order in Δt, just as for the arithmetic average and the centered finite difference approximation in (170). With the geometric mean, the resulting discrete equation

$$\frac{v^{n+1} - v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

becomes a *linear* equation in v^{n+1}, and we can therefore easily solve for v^{n+1}:

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,. \tag{172}$$

Using a geometric mean instead of an arithmetic mean in the Crank-Nicolson scheme is an attractive method for avoiding a nonlinear algebraic equation when discretizing a nonlinear ODE.

Physical data. Suitable values of μ are $1.8\cdot 10^{-5}$ Pa s for air and $8.9\cdot 10^{-4}$ Pa s for water. Densities can be taken as 1.2 kg/m^3 for air and as $1.0\cdot 10^3$ kg/m^3 for water. For considerable vertical displacement in the atmosphere one should take into account that the density of air varies with the altitude, see Section 9.8. One possible density variation arises from the one-layer model in the mentioned section.

Any density variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ in (172). To compute the density that enters $b^{n+\frac{1}{2}}$ we must also compute the vertical position $z(t)$ of the body. Since $v = dz/dt$, we can use a centered difference approximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

This $z^{n+\frac{1}{2}}$ is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then $b^{n+\frac{1}{2}}$.

The drag coefficient C_D depends heavily on the shape of the body. Some values are: 0.45 for a sphere, 0.42 for a semi-sphere, 1.05 for a cube, 0.82 for a long cylinder (when the center axis is in the vertical direction), 0.75 for a rocket, 1.0-1.3 for a man in upright position, 1.3 for a flat plate perpendicular to the flow, and 0.04 for a streamlined, droplet-like body.

Verification. To verify the program, one may assume a heavy body in air such that the F_b force can be neglected, and further assume a small velocity such that the air resistance F_d can also be neglected. This can be obtained by setting μ and ϱ to zero. The motion then leads to the velocity $v(t) = v_0 - gt$, which is linear in t and therefore should be reproduced to machine precision (say tolerance 10^{-15}) by any implementation based on the Crank-Nicolson or Forward Euler schemes.

Another verification, but not as powerful as the one above, can be based on computing the terminal velocity and comparing with the exact expressions. The advantage of this verification is that we can also test the situation $\varrho \neq 0$.

As always, the method of manufactured solutions can be applied to test the implementation of all terms in the governing equation, but then the solution has no physical relevance in general.

Scaling. Applying scaling, as described in Section 9.1, will for the linear case reduce the need to estimate values for seven parameters down to choosing one value of a single dimensionless parameter

$$\beta = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu I},$$

provided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled problem reads $\bar{u}' = 1 - \bar{u}$, $\bar{u}(0) = 0$, and there is no need for estimating physical parameters (!). This means that there is a single universal solution to the problem of a falling body starting from rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical cases correspond to stretching the $\bar{t}$ axis and the $\bar{u}$ axis in this dimensionless solution. More precisely, the physical velocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu} \bar{u}(t/(g(\varrho/\varrho_b - 1))) = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu} (1 - e^{t/(g(\varrho/\varrho_b - 1))})\,.$$

9.11 Decay ODEs from solving a PDE by Fourier expansions

Suppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

with boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = I(x)$. One may express the solution as

$$u(x,t) = \sum_{k=1}^{m} A_k(t) e^{ikx\pi/L},$$

for appropriate unknown functions A_k, $k = 1, \ldots, m$. We use the complex exponential $e^{ikx\pi/L}$ for easy algebra, but the physical u is taken as the real part of any complex expression. Note that the expansion in terms of $e^{ikx\pi/L}$ is compatible with the boundary conditions: all functions $e^{ikx\pi/L}$ vanish for $x = 0$ and $x = L$. Suppose we can express $I(x)$ as

$$I(x) = \sum_{k=1}^{m} I_k e^{ikx\pi/L}\,.$$

Such an expansion can be computed by well-known Fourier expansion techniques, but those details are not important here. Also, suppose we can express the given $f(x,t)$ as

$$f(x,t) = \sum_{k=1}^{m} b_k(t) e^{ikx\pi/L}\,.$$

Inserting the expansions for u and f in the differential equations demands that all terms corresponding to a given k must be equal. The calculations result in the follow system of ODEs:

$$A_k'(t) = -\alpha \frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1, \ldots, m\,.$$

From the initial condition

$$u(x,0) = \sum_k A_k(0) e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

so it follows that $A_k(0) = I_k$, $k = 1, \ldots, m$. We then have m equations of the form $A_k' = -aA_k + b$, $A_k(0) = I_k$, for appropriate definitions of a and b. These ODE problems are independent of each other such that we can solve one problem at a time. The outlined technique is a quite common solution approach to partial differential equations.

Remark. Since a_k depends on k and the stability of the Forward Euler scheme demands $a_k \Delta t \leq 1$, we get that $\Delta t \leq \alpha^{-1} L^2 \pi^{-2} k^{-2}$ for this scheme. Usually, quite large k values are needed to accurately represent the given functions I and f so that Δt in the Forward Euler scheme needs to be very small for these large values of k. Therefore, the Crank-Nicolson and Backward Euler schemes, which allow larger Δt without any growth in the solutions, are more popular choices when creating time-stepping algorithms for partial differential equations of the type considered in this example.

10 Exercises

Exercise 16: Radioactive decay of Carbon-14

The Carbon-14 isotope, whose radioactive decay is used extensively in dating organic material that is tens of thousands of years old, has a half-life of $5,730$ years. Determine the age of an organic material that contains 8.4 percent of its initial amount of Carbon-14. Use a time unit of 1 year in the computations. The uncertainty in the half time of Carbon-14 is ± 40 years. What is the corresponding uncertainty in the estimate of the age?

Hint 1. Let A be the amount of Carbon-14. The ODE problem is then $A'(t) = -aA(t)$, $A(0) = I$. Introduced the scaled amount $u = A/I$. The ODE problem for u is $u' = -au$, $u(0) = 1$. Measure time in years. Simulate until the first mesh point t_m such that $u(t_m) \leq 0.084$.

Hint 2. Use simulations with $5,730 \pm 40$ y as input and find the corresponding uncertainty interval for the result.

Solution. We need a tailored solver function for this exercise:

```
import numpy as np
import matplotlib.pyplot as plt

def solver(I, a, u_crit, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,t_m] until u <= u_crit
    with steps of dt. Return t_m.
    """
    # Use list for u and t since we do not know how many points
    # that are needed
    dt = float(dt)               # avoid integer division
    u = []
    t = []

    u.append(I)                  # assign initial condition
    t.append(0)
    while u[-1] > u_crit:
        u_new = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[-1]
        u.append(u_new)
        t.append(t[-1] + dt)
```

```
    return t[-1]


half_life = 5730
a  = np.log(2)/half_life
print 'Age:', solver(I=1, a=a, u_crit=0.084, dt=10, theta=0.5)
```

Running this code gives an age of 20,480 years.

The uncertainty can be estimated by the following code:

```
half_life_min = 5730 - 40
half_life_max = 5730 + 40
a_min = np.log(2)/half_life_min
a_max = np.log(2)/half_life_max
age_min = solver(I=1, a=a_max, u_crit=0.084, dt=10, theta=0.5)
age_max = solver(I=1, a=a_min, u_crit=0.084, dt=10, theta=0.5)
print 'Uncertainty: [%g, %g]' % (age_min, age_max)
```

Filename: `carbon14`.

Exercise 17: Derive schemes for Newton's law of cooling

Show in detail how we can apply the ideas of the Forward Euler, Backward Euler, and Crank-Nicolson discretizations to derive explicit computational formulas for new temperature values in Newton's law of cooling (see Section 9.4):

$$\frac{dT}{dt} = -k(T - T_s(t)), \quad T(0) = T_0\,.$$

Here, T is the temperature of the body, $T_s(t)$ is the temperature of the surroundings, t is time, k is the heat transfer coefficient, and T_0 is the initial temperature of the body. Summarize the discretizations in a θ-rule such that you can get the three schemes from a single formula by varying the θ parameter.

Solution. The idea of the Forward Euler scheme is to sample the ODE at $t = t_n$ and apply a forward difference approximation to the derivative:

$$\frac{T^{n+1} - T^n}{\Delta t} = -k(T^n - T_s(t_n))\,.$$

The Backward Euler applies a backward difference instead:

$$\frac{T^n - T^{n-1}}{\Delta t} = -k(T^n - T_s(t_n))\,.$$

The Crank-Nicolson scheme samples the ODE at $t_{n+\frac{1}{2}}$, applies a centered difference approximation, and an arithmetic mean approximation to $T^{n+\frac{1}{2}}$:

$$\frac{T^{n+1} - T^n}{\Delta t} = -k(T^{n+\frac{1}{2}} - T_s(t_{n+\frac{1}{2}})) \approx -k(\frac{1}{2}(T^n + T^{n+1}) - T_s(t_{n+\frac{1}{2}}))\,.$$

For each scheme we solve with respect to the unknown T^{n+1} (note that we switch index from n to $n+1$ in the Backward Euler scheme):

$$
\begin{aligned}
T^{n+1} &= T^n - k\Delta t(T^n - T_s(t_n)),\\
T^{n+1} &= \frac{T^n + k\Delta t T_s(t_{n+1})}{1 + k\Delta t},\\
T^{n+1} &= \frac{T^n - \frac{1}{2}k\Delta t T^n + k\Delta t T_s(t_{n+\frac{1}{2}})}{1 + \frac{1}{2}k\Delta t}\,.
\end{aligned}
$$

A θ scheme can be formulated as

$$
T^{n+1} = \frac{T^n - (1-\theta)k\Delta t T^n + k\Delta t T_s((1-\theta)t_n + \theta t_{n+1})}{1 + \theta k\Delta t}
$$

Filename: `schemes_cooling`.

Exercise 18: Implement schemes for Newton's law of cooling

The goal of this exercise is to implement the schemes from Exercise 17 and investigate several approaches for verifying the implementation.

a) Implement the θ-rule from Exercise 17 in a function

```
cooling(T0, k, T_s, t_end, dt, theta=0.5)
```

where `T0` is the initial temperature, `k` is the heat transfer coefficient, `T_s` is a function of `t` for the temperature of the surroundings, `t_end` is the end time of the simulation, `dt` is the time step, and `theta` corresponds to θ. The `cooling` function should return the temperature as an array `T` of values at the mesh points and the time mesh `t`.

Solution. Here is an appropriate function:

```
import numpy as np

def cooling(T0, k, T_s, t_end, dt, theta=0.5):
    """
    Solve T'=-k(T-T_s(t)), T(0)=T0,
    for t in (0,t_end] with steps of dt.
    T_s(t) is a Python function of t.
    theta=0.5 means Crank-Nicolson, 1 is Backward
    Euler, and 0 is Forward Euler scheme.
    """
    dt = float(dt)                  # avoid integer division
    Nt = int(round(t_end/dt))       # no of time intervals
    t_end = Nt*dt                   # adjust to fit time step dt
    T = np.zeros(Nt+1)              # array of T[n] values
    t = np.linspace(0, t_end, Nt+1) # time mesh
    T[0] = T0                       # set initial condition
```

```
    for n in range(0, Nt):          # n=0,1,...,Nt-1
        T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
        dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
        (1 + dt*theta*k)
    return T, t
```

b) In the case $\lim_{t\to\infty} T_s(t) = C = \text{const}$, explain why $T(t) \to C$. Construct an example where you can illustrate this property in a plot. Implement a corresponding test function that checks the correctness of the asymptotic value of the solution.

Solution. Apply the limit to the ODE:

$$\lim_{t\to\infty} \frac{dT}{dt} = -k(\lim_{t\to\infty} T - \lim_{t\to\infty} T_s)\,.$$

Assuming steady state behavior, $dT/dt \to \infty$ as $t \to \infty$. Then we get

$$0 = -k(\lim_{t\to\infty} T - C),$$

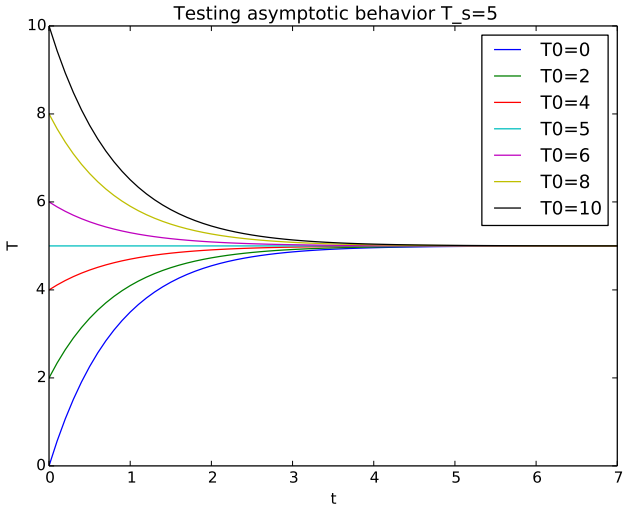
which means

$$\lim_{t\to\infty} T = C\,.$$

A corresponding test function takes the form

```
def test_asymptotic():
    """
    Test that ``any'' initial condition leads to
    the same asymptotic behavior when T_s=constant.
    """
    import matplotlib.pyplot as plt
    plt.figure()
    T_s = 5.
    k = 1.2
    dt = 0.1
    tol = 0.01  # tolerance for testing asymptotic value
    t_end = 7    # make sure t_end is large enough for tol
    T0_values = [0, 2, 4, 5, 6, 8, 10] # test many cases

    for T0 in [0, 2, 4, 5, 6, 8, 10]:
        u, t = cooling(T0, k, lambda t: T_s, t_end, dt)
        plt.plot(t, u)

        assert abs(u[-1] - T_s) < tol, '%s != %s' % (u[-1], T_s)

    plt.legend(['T0=%g' % T0 for T0 in T0_values])
    plt.title('Testing asymptotic behavior T_s=%g' % T_s)
    plt.xlabel('t')
    plt.ylabel('T')
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
    plt.show()
```

Note that we have added a plot in the test function for convenience. Letting test functions perform plotting is, however, not a good idea if you want to run a large set of tests.



c) A piecewise constant surrounding temperature,

$$T_s(t) = \begin{cases} C_0, & 0 \leq t \leq t^* \\ C_1, & t > t^*, \end{cases}$$

corresponds to a sudden change in the environment at $t = t^*$. Choose $C_0 = 2T_0$, $C_1 = \frac{1}{2}T_0$, and $t^* = 3/k$. Plot the solution $T(t)$ and explain why it seems physically reasonable.

Solution. First we implement a general tool for piecewise constant functions:

```
class Piecewise(object):
    """Class for holding a piecewise constant function."""
    def __init__(self, C0, C1, t_star):
        self.C0, self.C1 = C0, C1
        self.t_star = t_star

    def __call__(self, t):
        """
        Return value of piecewise constant function.
        t can be float or numpy array.
        """
        if isinstance(t, (float,int)):
            if t <= self.t_star:
                T_s = self.C0
            elif t > self.t_star:
                T_s = self.C1
        else:
            # assume numpy array
            T_s = np.piecewise(t,
```

```
                                [t <= self.t_star, t > self.t_star],
                                [self.C0, self.C1])
        # Alternative
        # T_s = np.where(t <= self.t_star, C0, C1)
    return T_s
```

It is convenient to scale the problem such that we do not need to find physically relevant values for k. A common scaling of T is

$$\bar{T} = \frac{T - T_0}{T_s - T_0},$$

when T_s is constant since then $\bar{T} \in [0, 1]$. Here, we may choose the long-term value of T_s in the denominator such that $\lim_{t\to\infty} \bar{T} = 1$, i.e.,

$$\bar{T} = \frac{T - T_0}{0.5T_0 - T_0} = -2\frac{T - T_0}{T_0},$$

but it leads to a shift in the sign of the temperature on the right-hand side of the ODE, and we cannot reuse the code for the original problem in the dimensionless case. We therefore avoid the negative sign and use a temperature scale $2T_0 - T_0$,

$$\bar{T} = \frac{T - T_0}{2T_0 - T_0} = \frac{T - T_0}{T_0},$$

which gives $\bar{T}$ varying from 0 initially to a maximum of 1 and finally to a minimum of $-\frac{1}{2}$. We scale T_s by its maximum value $2T_0$ so $\bar{T}_s \in [0, 1]$:

$$\bar{T}_s(\bar{t}) = \frac{T_s(t)}{\max_t T_s(t)} = \frac{T_s(t_c\bar{t})}{2T_0} = \begin{cases} 1, & \bar{t} < t^*/t_c, \\ \frac{1}{4}, & \bar{t} \geq t^*/t_c \end{cases}$$

where t_c is the time scale. Inserted in the ODE we get

$$\frac{T_0}{t_c}\frac{d\bar{T}}{d\bar{t}} = -k(T_0\bar{T} + T_0 - 2T_0\bar{T}_s,$$

leading to

$$\frac{d\bar{T}}{d\bar{t}} = -kt_c(\bar{T} + 1 - 2\bar{T}_s)\,.$$

A natural choice is $t_c = 1/k$ so we get the scaled problem

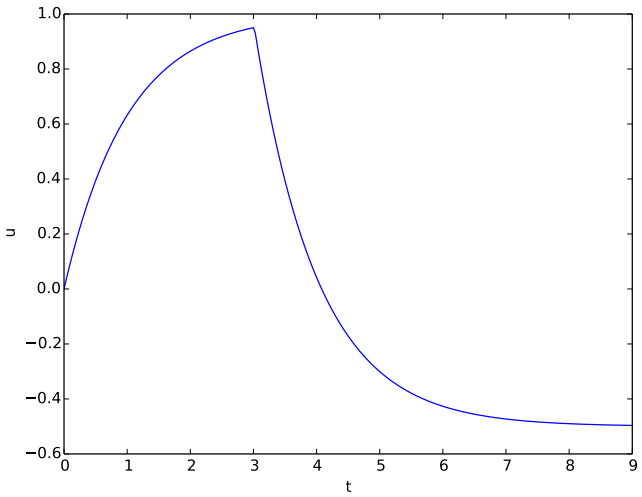
$$\frac{d\bar{T}}{d\bar{t}} = -(\bar{T} + 1 - 2\bar{T}_s) = -(\bar{T} - (2\bar{T}_s - 1)), \quad \bar{T}(0) = 0\,.$$

We can simulate this problem using the code for the original problem by choosing $k = 1$, $T_0 = 0$, and $T_s = (2 - 1) = 1$ for $t < 3$ and $T_s = (2\frac{1}{4} - 1) = -\frac{1}{2}$ for $t > 3$.

The appropriate code becomes

```
def simulate_piecewise_constant_Ts():
    """
    Simulate scaled problem: T' = -(T - (2T_s-1)), T(0)=0,
    where T_s=1 for t < 3 and -0.5 for t > 3.
    """
    k = 1
    T0 = 0
    t_star = 3.0
    C0 = 1
    C1 = -0.5
    T_s = Piecewise(C0, C1, t_star)
    dt = t_star/100.0
    T, t = cooling(T0, k, T_s, t_end=3*t_star, dt=dt, theta=0.5)
    import matplotlib.pyplot as plt
    plt.figure()
    plt.plot(t, T)
    plt.xlabel('t');  plt.ylabel('u')
    plt.savefig('tmp2.png');  plt.savefig('tmp2.pdf')
    plt.show()
```

The plot looks like this:



The result is reasonable because first $T_s = 1$ and the body's temperature will try to rise from 0 to 1, and it almost gets there in the time $[0, 3]$, before $T_s = -0.5$ and then the body is cooled down to -0.5 as t increases, and this is also the asymptotic value.

d) We know from the ODE $u' = -au$ that the Crank-Nicolson scheme can give non-physical oscillations for $\Delta t > 2/a$. In the present problem, this results indicates that the Crank-Nicolson scheme give undesired oscillations for $\Delta t > 2/k$. Discuss if this a potential problem in the physical case from c).

Solution. In the unscaled problem, the first stage of the simulation is covers time interval $[0, t^*] = [0, 3/k]$. It makes sense to choose Δt significantly smaller

than $3/k$, and the stability limit $2/k$ is a too large step. The next time level is then $4/k$, and it sounds reasonable to include the point $t^* = 3/k$ as a mesh point. Oscillations would then occur if we choose $\Delta t = 3/k$, but this means only one step through the first interval $[0, t^*]$, which is a very coarse mesh. Halving Δt is still a coarse mesh, but then there cannot be oscillations.

e) Find an expression for the exact solution of $T' = -k(T - T_s(t))$, $T(0) = T_0$. Construct a test case and compare the numerical and exact solution in a plot.

Find a value of the time step Δt such that the two solution curves cannot (visually) be distinguished from each other. Many scientists will claim that such a plot provides evidence for a correct implementation, but point out why there still may be errors in the code. Can you introduce bugs in the `cooling` function and still achieve visually coinciding curves?

Hint. The exact solution can be derived by multiplying (134) by the integrating factor e^{kt}.

Solution. Multiplication of e^{kt}, using the product rule for differentiation "backwards", and integrating from 0 to t, results in

$$\int_0^t (e^{kt}T)'dt = k\int_0^t e^{kt}T_s dt\,.$$

The left-hand side becomes $e^{kt}T(t) - T_0$. Multiplying by e^{-kt} then gives

$$T(t) = T_0 e^{-kt} + ke^{-kt}\int_0^t e^{k\tau}T_s(\tau)d\tau,$$

which is the general expression for the exact solution.

As a check, we consider the case where T_s is constant. That problem can easily be solved by introducing $u = T - T_s$, resulting in $u' = -ku$, $u(0) = T_0 - T_s$, with solution $u(t) = (T_0 - T_s)e^{-kt}$, and consequently $T = T_s + (T_0 - T_s)e^{-kt}$. With a constant T_s in the general solution above, the solution becomes

$$\begin{aligned} T(t) &= T_0 e^{-kt} + ke^{-kt}\int_0^t e^{kt}T_s dt \\ &= T_0 e^{-kt} + ke^{-kt}T_s k^{-1}(e^{kt} - 1) \\ &= T_0 e^{-kt} + T_s - T_s e^{-kt} \\ &= T_s + (T_0 - T_s)e^{-kt}, \end{aligned}$$

as desired.

We choose the same test problem as in c) and use SymPy to do the integration. A function doing the integration and returning Python functions for the formulas for $t < t^*$ and $t \geq t^*$ is convenient:

```
def symbolic_exact_solution(verbose=False):
    """Compute the exact solution formula via sympy."""
    # sol1: solution for t < t_star,
    # sol2: solution for t > t_star
    import sympy as sym
    T0 = sym.symbols('T0')
    k = sym.symbols('k', positive=True)
    # Piecewise linear T_sunction
    t, t_star, C0, C1 = sym.symbols('t t_star C0 C1')
    T_s = C0
    I = sym.integrate(sym.exp(k*t)*T_s, (t, 0, t))
    sol1 = T0*sym.exp(-k*t) + k*sym.exp(-k*t)*I
    sol1 = sym.simplify(sym.expand(sol1))
    if verbose:
        # Some debugging print
        print 'solution t < t_star:', sol1
        #print sym.latex(sol1)
    T_s = C1
    I = sym.integrate(sym.exp(k*t)*C0, (t, 0, t_star)) + \
        sym.integrate(sym.exp(k*t)*C1, (t, t_star, t))
    sol2 = T0*sym.exp(-k*t) + k*sym.exp(-k*t)*I
    sol2 = sym.simplify(sym.expand(sol2))
    if verbose:
        print 'solution t > t_star:', sol2
        #print sym.latex(sol2)

    # Convert to numerical functions
    exact0 = sym.lambdify([t, C0, k, T0],
                          sol1, modules='numpy')
    exact1 = sym.lambdify([t, C0, C1, t_star, k, T0],
                          sol2, modules='numpy')
    return exact0, exact1
```

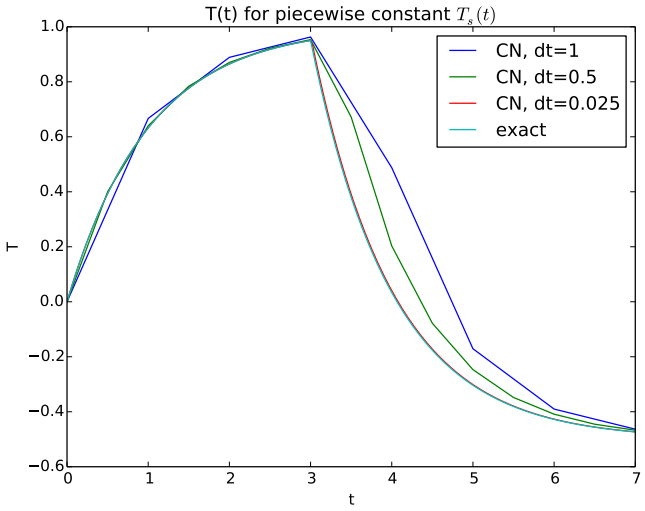
Then we need a function that can evaluate the exact solution as a mesh function:

```
def evaluate_exact_solution(t, k, T0, C0, C1, t_star,
                            verbose=False):
    """
    Return exact (analytical) solution of the problem.
    Exact solution is produced by sympy.
    """
    exact0, exact1 = symbolic_exact_solution()
    # exact0/1 works with t as numpy array
    if isinstance(t, (float,int)):
        if t < t_star:
            return exact0(t, C0, k, T0)
        else:
            return exact1(t, C0, C1, t_star, k, T0)
    else:
        # assume numpy array
        return np.where(
            t < t_star,
            exact0(t, C0, k, T0),
            exact1(t, C0, C1, t_star, k, T0))
```

Finally we can run the comparison:

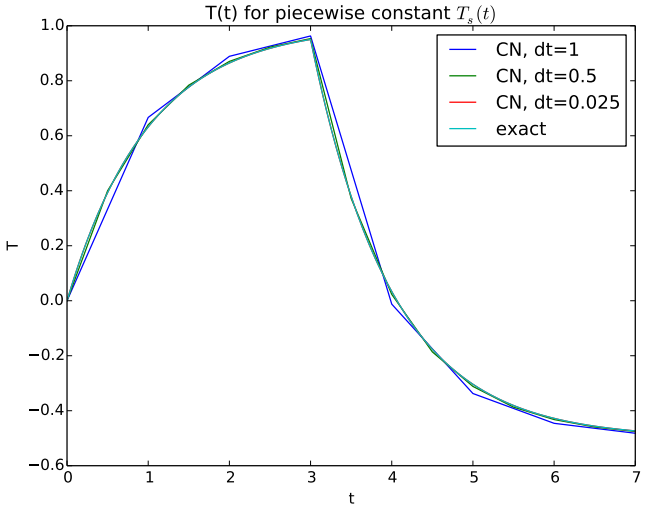
```
def compare_numerical_and_exact_solution():
    """
    Compare exact and numerical solution with piecewise
    constant surrounding temperature. Use scaled problem
    from function simulate_piecewise_constant_Ts.
    """
    T0 = 0
    k = 1
    C0 = 1
    C1 = -0.5
    t_star = 3
    t_end = 7

    T_s = Piecewise(C0, C1, t_star)

    import matplotlib.pyplot as plt
    plt.figure()
    dt_values = [1, 0.5, 0.025]
    #dt_values = [0.025]
    for dt in dt_values:
        T, t = cooling(T0, k, T_s, t_end, dt, theta=0.5)
        plt.plot(t, T)

    t_e = np.linspace(0, t_end, 1001)  # find mesh for T_e
    # Could use sym.Rational(1,2) instead of 0.5, but not necessary
    # when we are not interested in symbolic formulas
    T_e = evaluate_exact_solution(t_e, k, T0, C0, C1, t_star)
    plt.plot(t_e, T_e)
    plt.legend(['CN, dt=%g' % dt for dt in dt_values] + ['exact'])
    plt.title('T(t) for piecewise constant $T_s(t)$')
    plt.xlabel('t')
    plt.ylabel('T')
    plt.savefig('tmp3.png');  plt.savefig('tmp3.pdf')
    plt.show()
```

The Δt values were found after some trial and error, but they illustrate crude approximations and one with the biggest possible Δt such that the exact solution and the numerical solution cannot be visually distinguished:

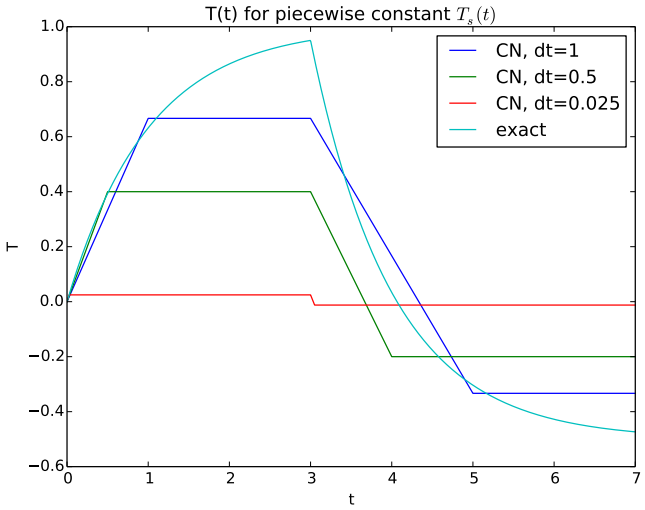


We can now start to introduce bugs in the `cooling` function to see if it is possible to have some Δt and still find coinciding curves.

Bug 1: Wrong time level in the T_s function. We replace `T_s[n]` by `T_s[n+1]` in the implementation of the scheme and rerun the case. Now the lowest Δt is still on top of the exact solution, but the numerical solution on the two coarser meshes are more accurate! This is because we lower the surrounding temperature somewhat earlier in the buggy scheme and this reduces the "overshoot" on the coarsest meshes in the figure above.



Bug 2: Wrong time level in the T function. We can replace `T[n]` by `T[n+1]` on the right-hand side of the scheme. This is a serious error since `T[n+1]` is not yet computed and therefore equal to zero when `T` was made by calling `np.zeros`. The results are also nonsense, and one would immediately look for a bug.



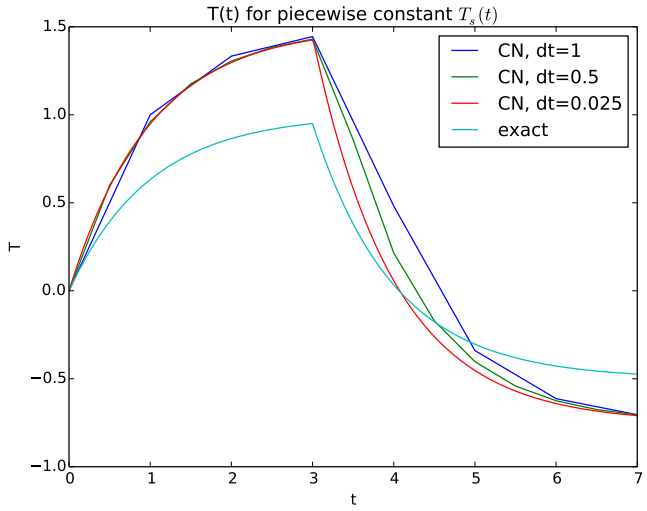
Bug 3: Missing θ in a term. Let us forget to multiply by `theta` in the nominator of the scheme, i.e., we replace

```
T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
         dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
         (1 + dt*theta*k)
```

by

```
T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
         dt*k*(T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
         (1 + dt*theta*k)
```

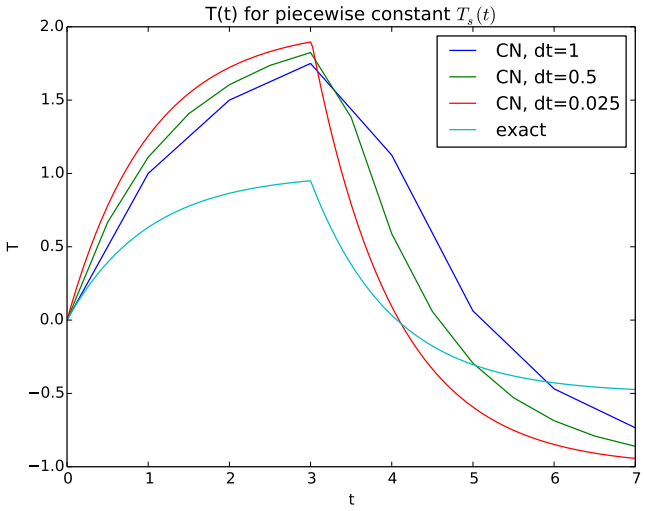
This error leads to convergence towards a wrong solution:



If we did not have the exact solution, one could be led to think that the solution was correct, but it is non-physical since we do not expect the temperature to rise from T_0 to a level *above* the surrounding temperature. The plot shows that $\bar{T} > 1$, the value of the (scaled) surrounding temperature. Note that if we used the Backward Euler scheme instead of the Crank-Nicolson scheme, this bug would have no effect!

Bug 4: Missing `k` in the updating formula. Obviously, when we solve the scaled problem where $k = 1$ by definition, such a programming mistake has no effect. Otherwise, k influences the time scale, so there will be a stretch of the time axis in the numerical solution and this should be easily detected in a plot.

Bug 5: Using `1-theta` instead of `theta`. Such an error is not detectable in the Crank-Nicolson scheme, but will have a significant effect in the other schemes. As a test, we replace `1-theta` in the nominator by `theta`. This leads to $T = 0$ in the Forward Euler scheme, but a reasonable shape in the Backward Euler scheme, although the solution becomes larger than the surrounding temperature (1 in the scaled problem).



f) Implement a test function for checking that the solution returned by the `cooling` function is identical to the exact numerical solution of the problem (to machine precision) when T_s is constant.

Hint. The exact solution of the discrete equations in the case T_s is a constant can be found by introducing $u = T - T_s$ to get a problem $u' = -ku$, $u(0) = T_0 - T_s$. The solution of the discrete equations is then of the form $u^n = (T_0 - T_s)A^n$ for some amplification factor A. The expression for T^n is then $T^n = T_s(t_n) + u^n = T_s + (T_0 - T_s)A^n$. We find that

$$A = \frac{1 - (1-\theta)k\Delta t}{1 + \theta k \Delta t}\,.$$

The test function, testing several θ values for a quite coarse mesh, may take the form

```
def test_discrete_solution():
    """
    Compare the numerical solution with an exact solution of the scheme
    when the T_s is constant.
    """
    T_s = 10
    T0 = 2
    k = 1.2
    dt = 0.1   # can use any mesh
    N_t = 6    # any no of steps will do
    t_end = dt*N_t
    t = np.linspace(0, t_end, N_t+1)

    for theta in [0, 0.5, 1, 0.2]:
        u, t = cooling(T0, k, lambda t: T_s , t_end, dt, theta)
        A = (1 - (1-theta)*k*dt)/(1 + theta*k*dt)
        u_discrete_exact = T_s + (T0-T_s)*A**(np.arange(len(t)))
```

```
        diff = np.abs(u - u_discrete_exact).max()
        print 'diff computed and exact discrete solution:', diff
        tol = 1E-14
        success = diff < tol
        assert success, 'diff=%g' % diff
```

Running this function shows that the `diff` variable is `3.55E-15` as maximum so a tolerance of 10^{-14} is appropriate. This is a good test that the `cooling` function works!
Filename: `cooling`.

Exercise 19: Find time of murder from body temperature

A detective measures the temperature of a dead body to be 26.7 C at 2 pm. One hour later the temperature is 25.8 C. The question is when death occurred.

Assume that Newton's law of cooling (134) is an appropriate mathematical model for the evolution of the temperature in the body. First, determine k in (134) by formulating a Forward Euler approximation with one time steep from time 2 am to time 3 am, where knowing the two temperatures allows for finding k. Assume the temperature in the air to be 20 C. Thereafter, simulate the temperature evolution from the time of murder, taken as $t = 0$, when $T = 37$ C, until the temperature reaches 25.8 C. The corresponding time allows for answering when death occurred.

Solution. A Forward Euler step from T^0 to T^1 reads

$$T^1 = T_0 + -k\Delta t(T^0 - T_s),$$

and solving with respect to k results in

$$k = \frac{T^1 - T^0}{\Delta t(T_s - T_0)}\,.$$

We implement this formula in a function,

```
def estimate_k(T0, T1, Ts, dt):
    return float(T1 - T0)/(dt*(Ts - T0))
```

We have $T_0 = 26.7$ C, $T_1 = 25.8$ C, $T_s = 20$ C, and $\Delta t = 1$ h, i.e., $\Delta t = 3600$ s. The proper call is therefore

```
k = estimate_k(26.7, 25.8, 20, 3600)
```

For the simulation we use the Forward Euler method,

$$T^{n+1} = T^n - k\Delta t(T^n - T_s),$$

and simulate as long as $T > 25.8$ C:

```
T = 37
Ts = 20
from cooling import cooling
while T > 25.8:
    T = T - k*dt*(T - Ts)
    t+= dt

minutes, seconds = divmod(t, 60)
hours, minutes = divmod(minutes, 60)
print """
The death occurred %d hours, %d minutes,
and %g seconds before 3am.""" % (hours, minutes, seconds)
```

The result of running the code becomes

```
Terminal> python detective.py
k=3.73134e-05

The death occurred 8 hours, 0 minutes,
and 19 seconds before 3am.
```

Filename: `detective`.

Exercise 20: Simulate an oscillating cooling process

The surrounding temperature T_s in Newton's law of cooling (134) may vary in time. Assume that the variations are periodic with period P and amplitude a around a constant mean temperature T_m:

$$T_s(t) = T_m + a\sin\left(\frac{2\pi}{P}t\right). \tag{173}$$

Simulate a process with the following data: $k = 0.05$ min^{-1}, $T(0) = 5$ C, $T_m = 25$ C, $a = 2.5$ C, and $P = 1$ h, $P = 10$ min, and $P = 6$ h. Plot the T solutions and T_s in the same plot.

Solution. We can reuse the `cooling` function from Exercise 18 to do the simulations:

```
import numpy as np

def cooling(T0, k, T_s, t_end, dt, theta=0.5):
    """
    Solve T'=-k(T-T_s(t)), T(0)=T0,
    for t in (0,t_end] with steps of dt.
    T_s(t) is a Python function of t.
    theta=0.5 means Crank-Nicolson, 1 is Backward
    Euler, and 0 is Forward Euler scheme.
    """
    dt = float(dt)                  # avoid integer division
    Nt = int(round(t_end/dt))       # no of time intervals
    t_end = Nt*dt                   # adjust to fit time step dt
```

```
    T = np.zeros(Nt+1)              # array of T[n] values
    t = np.linspace(0, t_end, Nt+1) # time mesh
    T[0] = T0                       # set initial condition
    for n in range(0, Nt):          # n=0,1,...,Nt-1
        T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
        dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
        (1 + dt*theta*k)
    return T, t
```

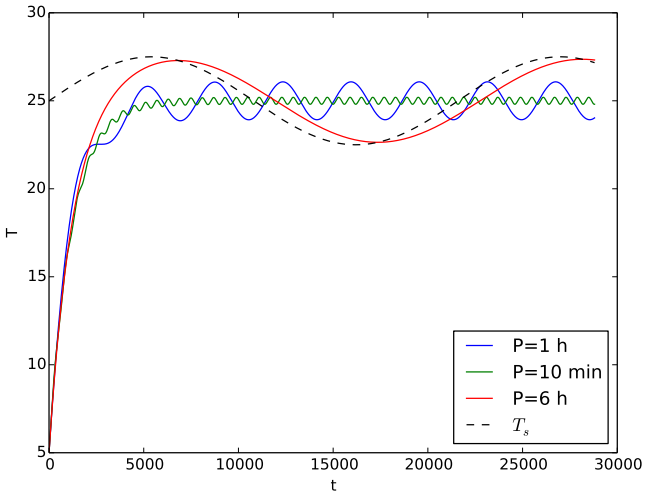
The challenge is to use the right units for the input data. We can use Celsius for temperature since it has the same increments as Kelvin. Time quantities should be measured in seconds:

$$k = 20 \text{ min}^{-1} = \frac{20}{60} \text{ s}^{-1},$$
$$P = (1 \text{ h} = 3600 \text{ s}, 10 \text{ min} = 600 \text{ s}, 6 \text{ h} = 6 \cdot 3600 \text{ s}) .$$

To achieve reasonable accuracy, we choose Δt as 40 steps per the shortest period of the T_s oscillations: $\Delta t = 600/40$. With some trials we find an appropriate simulation interval for all three cases to be $[0, 8]$ h.

The code becomes

```
from cooling import cooling
from numpy import pi, sin

def T_s(t):
    return Tm + a*sin((2*pi/P)*t)

Tm = 25
a = 2.5
P_values = [3600, 600, 3600*6]
k = 0.05/60
T0 = 5

import matplotlib.pyplot as plt
for P in P_values:
    T, t = cooling(T0, k, T_s, t_end=8*3600, dt=600/40)
    plt.plot(t, T)
plt.plot(t, T_s(t), 'k--')
legends = ['P=1 h', 'P=10 min', 'P=6 h', '$T_s$']
plt.legend(legends, loc='lower right')
plt.xlabel('t'); plt.ylabel('T')
plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
plt.show()
```



Discussion of the results. We see that it takes some time to increase the temperature from T_0 to oscillations around T_m. When T_s oscillates fast ($P = 10$ min), k is not large enough so that T can reach the surrounding temperature in the time available before the surrounding temperature decreases. However, for large P (6 h), there is almost enough time to heat and cool the object to reach the maximum and minimum temperatures of the surroundings.
Filename: `osc_cooling`.

Exercise 21: Simulate stochastic radioactive decay

The purpose of this exercise is to implement the stochastic model described in Section 9.5 and show that its mean behavior approximates the solution of the corresponding ODE model.

The simulation goes on for a time interval $[0, T]$ divided into N_t intervals of length Δt. We start with N_0 atoms. In some time interval, we have N atoms that have survived. Simulate N Bernoulli trials with probability $\lambda\Delta t$ in this interval by drawing N random numbers, each being 0 (survival) or 1 (decay), where the probability of getting 1 is $\lambda\Delta t$. We are interested in the number of decays, d, and the number of survived atoms in the next interval is then $N - d$. The Bernoulli trials are simulated by drawing N uniformly distributed real numbers on $[0, 1]$ and saying that 1 corresponds to a value less than $\lambda\Delta t$:

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
Bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = Bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true and false values become 1 and 0, respectively, when converted to an integer array.

Repeat the simulation over $[0, T]$ a large number of times, compute the average value of N in each interval, and compare with the solution of the corresponding ODE model. Filename: `stochastic_decay`.

Exercise 22: Radioactive decay of two substances

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a half-life $A_{1/2}$, while substance B decay to form type A nuclei with a half-life $B_{1/2}$. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u'_A = u_B/B_{1/2} - u_A/A_{1/2}, \tag{174}$$

$$\frac{1}{\ln 2}u'_B = u_A/A_{1/2} - u_B/B_{1/2}, \tag{175}$$

with $u_A(0) = u_B(0) = 1$.

a) Make a simulation program that solves for $u_A(t)$ and $u_B(t)$.

b) Verify the implementation by computing analytically the limiting values of u_A and u_B as $t \to \infty$ (assume $u'_A, u'_B \to 0$) and comparing these with those obtained numerically.

c) Run the program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$ minutes. Use a time unit of 1 minute. Plot u_A and u_B versus time in the same plot. Filename: `radioactive_decay_2subst`.

Exercise 23: Simulate a simple chemical reaction

Consider the simple chemical reaction where a substance A is turned into a substance B according to

$$\frac{[A]}{dt} = -k[A],$$

$$\frac{[B]}{dt} = k[A],$$

where $[A]$ and $[B]$ are the concentrations of A and B, respectively. It may be a challenge to find appropriate values of k, but we can avoid this problem by working with a scaled model (as explained in Section 9.1). Scale the model above, using a time scale $1/k$, and use the initial concentration of $[A]$ as scale for $[A]$ and $[B]$. Show that the scaled system reads

$$\frac{u}{dt} = -u,$$
$$\frac{v}{dt} = u,$$

with initial conditions $u(0) = 1$, and $v(0) = \alpha$, where $\alpha = [B](0)/[A](0)$ is a dimensionless number, and u and v are the scaled concentrations of $[A]$ and $[B]$, respectively. Implement a numerical scheme that can be used to find the solutions $u(t)$ and $v(t)$. Visualize u and v in the same plot. Filename: `chemcial_kinetics_AB`.

Exercise 24: Simulate an n-th order chemical reaction

An n-order chemical reaction, generalizing the model in Exercise 23, takes the form

$$\frac{[A]}{dt} = -k[A]^n,$$
$$\frac{[B]}{dt} = k[A]^n,$$

where symbols are as defined in Exercise 23. Bring this model on dimensionless form, using a time scale $[A](0)^{n-1}/k$, and show that the dimensionless model simplifies to

$$\frac{u}{dt} = -u^n,$$
$$\frac{v}{dt} = u^n,$$

with $u(0) = 1$ and $v(0) = \alpha = [B](0)/[A](0)$. Solve numerically for $u(t)$ and show a plot with u for $n = 0.5, 1, 2, 4$. Filename: `chemcial_kinetics_ABn`.

Exercise 25: Simulate spreading of a disease

The SIR model (153)-(155) can be used to simulate spreading of an epidemic disease.

a) Estimating the parameter β is difficult so it can be handy to scale the equations. Use $t_c = 1/\nu$ as time scale, and scale S, I, and R by the population size $N = S(0) + I(0) + R(0)$. Show that the resulting dimensionless model becomes

$$\frac{d\bar{S}}{d\bar{t}} = -R_0\bar{S}\bar{I}, \tag{176}$$

$$\frac{d\bar{I}}{d\bar{t}} = R_0\bar{S}\bar{I} - \bar{I}, \tag{177}$$

$$\frac{d\bar{R}}{d\bar{t}} = I, \tag{178}$$

$$\bar{S}(0) = 1 - \alpha, \tag{179}$$

$$\bar{I}(0) = \alpha, \tag{180}$$

$$\bar{R}(0) = 0, \tag{181}$$

where R_0 and α are the only parameters in the problem:

$$R_0 = \frac{N\beta}{\nu}, \quad \alpha = \frac{I(0)}{N}\,.$$

Solution. We introduce

$$\bar{t} = \frac{t}{\nu^{-1}}, \quad \bar{S} = \frac{S}{N}, \quad \bar{I} = \frac{I}{N}, \quad \bar{R} = \frac{R}{N}\,.$$

Inserting these expressions in the governing equations and dividing by νN gives the listed dimensionless ODEs. The scaled initial condition for $\bar{S}(0)$ follows from $\bar{S}(0) = S(0)/N = (N - I(0))/N = 1 - \alpha$, since initially, $R(0) = 0$ and therefore $N = S(0) + I(0)$.

b) Show that the R_0 parameter governs whether the disease will spread or not at $t = 0$.

Hint. Spreading means $dI/dt > 0$.

Solution. For dI/dt to be positive, we must have $(R_0\bar{S}-1)\bar{I} > 0$, i.e., $R_0\bar{S}-1 > 0$ since $\bar{I} \geq 0$. At $t = 0$, we get $R_0\bar{S}(0) > 1$ as the criterion, or

$$\tilde{R}_0\bar{S}(0) = \frac{N\beta}{\nu}\frac{S(0)}{N} = \frac{S(0)\beta}{\nu} > 1\,.$$

The dimensionless parameter $S(0)\beta/\nu$ is denoted by R_0 in the epidemiology literature and known as the *basic reproductive number*.

c) Implement the scaled SIR model. Check at every time step, as a verification, that $\bar{S} + \bar{I} + \bar{R} = 1$.

d) Simulate the spreading of a disease where $R_0 = 1.1$ and 1 percent of the population is infected at time $t = 0$.

Solution. The given data means that $\bar{I}(0) = \alpha = 0.01$ and $\bar{S}(0) = 0.99$.
Filename: `SIR`.

Exercise 26: Simulate a biochemical process

The purpose of this exercise is to simulate the ODE system (149)-(152) modeling a simple biochemical process.

a) Scale (149)-(152) such that we can work with dimensionless parameters, which are easier to prescribe. Introduce

$$\bar{Q} = \frac{[ES]}{Q_c}, \quad \bar{P} = \frac{P}{P_c}, \quad \bar{S} = \frac{S}{S_0}, \quad \bar{E} = \frac{E}{E_0}, \quad \bar{t} = \frac{t}{t_c},$$

where appropriate scales are

$$Q_c = \frac{S_0 E_0}{K}, \quad P_c = Q_c, \quad t_c = \frac{1}{k_+ E_0},$$

with $K = (k_v + k_-)/k_+$ is the Michaelis constant. Show that the scale system becomes

$$\frac{d\bar{Q}}{d\bar{t}} = \alpha(\bar{E}\bar{S} - \bar{Q}), \tag{182}$$

$$\frac{d\bar{P}}{d\bar{t}} = \beta\bar{Q}, \tag{183}$$

$$\frac{d\bar{S}}{d\bar{t}} = -\bar{E}\bar{S} + (1 - \beta\alpha^{-1})\bar{Q}, \tag{184}$$

$$\epsilon\frac{d\bar{E}}{d\bar{t}} = -\bar{E}\bar{S} + \bar{Q}, \tag{185}$$

where we have three dimensionless parameters

$$\alpha = \frac{K}{E_0}, \quad \beta = \frac{k_v}{k_+ E_0}, \quad \epsilon = \frac{E_0}{S_0}\,.$$

The corresponding initial conditions are $\bar{Q} = \bar{P} = 0$ and $\bar{S} = \bar{E} = 1$.

Solution. Replacing the unknowns and t by their dimensionless equivalents leads to

$$\begin{aligned}
\frac{d\bar{Q}}{d\bar{t}} &= t_c k_+ \frac{E_0 S_0}{Q_c}\bar{E}\bar{S} - t_c(k_v + k_-)\bar{Q}, \\
\frac{d\bar{P}}{d\bar{t}} &= t_c k_v \frac{Q_c}{P_c}\bar{Q}, \\
\frac{d\bar{S}}{d\bar{t}} &= -t_c k_+ E_0 \bar{E}\bar{S} + t_c k_- \frac{Q_c}{S_0}\bar{Q}, \\
\frac{d\bar{E}}{d\bar{t}} &= -t_c k_+ S_0 \bar{E}\bar{S} + t_c(k_- + k_v)\frac{Q_c}{E_0}\bar{Q}\,.
\end{aligned}$$

Inserting the choice of scales brings us to the given equations, after quite some algebra and identifying coefficients in terms of the provided dimensionless numbers.

b) Implement a function for solving (182)-(185).

Solution. Let us use Odespy to solve the differential equations, although a plain Forward Euler scheme will be fine.

```
import odespy
import numpy as np
import matplotlib.pyplot as plt
import sys

def solver(f, alpha, beta, epsilon, T, dt=0.1):
    def f(u, t):
        Q, P, S, E = u
        return [
            alpha*(E*S - Q),
            beta*Q,
            -E*S + (1-beta/alpha)*Q,
            (-E*S + Q)/epsilon,
            ]

    Nt = int(round(T/dt))
    t_mesh = np.linspace(0, Nt*dt, Nt+1)

    solver = odespy.RK4(f)
    solver.set_initial_condition([0, 0, 1, 1])
    u, t = solver.solve(t_mesh)
    Q = u[:,0]
    P = u[:,1]
    S = u[:,2]
    E = u[:,3]
    return Q, P, S, E
```

c) There are two conservation equations implied by (149)-(152):

$$[ES] + [E] = E_0, \tag{186}$$

$$[ES] + [S] + [P] = S_0\,. \tag{187}$$

Derive these two equations. Use these properties in the function in b) to do a partial verification of the solution at each time step.

Solution. Adding (149) and (152) shows that

$$\frac{d[ES]}{dt} + \frac{d[E]}{dt} = 0,$$

and therefore $[ES] + [E] = \text{const}$. Since $[ES](0) = 0$ and $[E](0) = E_0$, the constant is E_0 at $t = 0$ and will remain so. Similarly, adding (149), (151),

and (150) shows that their time derivatives sum up to zero, and therefore $[ES]+[S]+[P] = \text{const}$. Since $[P](0) = 0$, the constant must be $0+S_0+0 = S_0$.

To use the conservation as a consistency check in the software, we need to find the equivalent dimensionless versions:

$$[ES] + [E] = E_0 \quad \Rightarrow \quad Q_c\bar{Q} + E_0\bar{E} = E_0,$$

and from this we get, after a little algebra,

$$\alpha^{-1}\epsilon^{-1}\bar{Q} + \bar{E} = 1\,.$$

The other conservation equation becomes

$$\bar{Q} + \alpha\bar{S} + \bar{P} = \alpha\,.$$

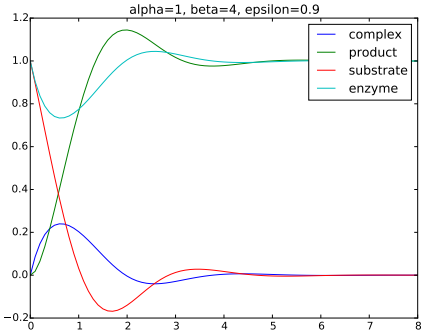
The implementation may go like

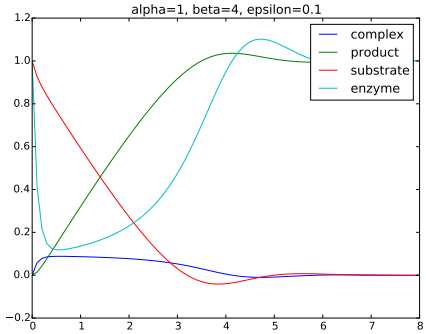
```
computed = Q[n+1]/(alpha*epsilon) + E[n+1]
expected = 1
diff1 = abs(computed - expected)

computed = Q[n+1] + alpha*S[n+1] + P[n+1]
expected = alpha
diff2 = abs(computed - expected)

tol = 1E-14
if diff1 < tol or diff2 < tol:
    print '*** Consistency check failed:', diff1, diff2
```

d) Simulate a case with $T = 8$, $\alpha = 1$, $\beta = 4$, and two ϵ values: 0.9 and 0.1.

```
def demo():
    alpha = 1
    beta = 4
    epsilon = 0.1
    T = 8
    dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.1
    Q, P, S, E = solver(alpha, beta, epsilon, T, dt)
    plt.plot(t, Q, t, P, t, S, t, E)
    plt.legend(['complex', 'product', 'substrate', 'enzyme'],
               loc='upper right')
    plt.title('alpha=%g, beta=%g, epsilon=%g' % (alpha, beta, epsilon))
    plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
    plt.show()

if __name__ == '__main__':
    demo()
```

Solution.





Filename: `biochem`.

Exercise 27: Simulate the pressure drop in the atmosphere

We consider the models for atmospheric pressure in Section 9.8. Make a program with three functions,

- one computing the pressure $p(z)$ using a seven-layer model and varying L,
- one computing $p(z)$ using a seven-layer model, but with constant temperature in each layer, and
- one computing $p(z)$ based on the one-layer model.

How can these implementations be verified? Should ease of verification impact how you code the functions? Compare the three models in a plot. Filename: `atmospheric_pressure`.

Exercise 28: Make a program for vertical motion in a fluid

Implement the Stokes' drag model (164) and the quadratic drag model (167) from Section 9.10, using the Crank-Nicolson scheme and a geometric mean for $|v|v$ as explained, and assume constant fluid density. At each time level, compute the Reynolds number Re and choose the Stokes' drag model if $\text{Re} < 1$ and the quadratic drag model otherwise.

The computation of the numerical solution should take place either in a stand-alone function or in a solver class that looks up a problem class for physical data. Create a module and equip it with pytest/nose compatible test functions for automatically verifying the code.

Verification tests can be based on

- the terminal velocity (see Section 9.10),
- the exact solution when the drag force is neglected (see Section 9.10),
- the method of manufactured solutions (see Section 6.5) combined with computing convergence rates (see Section 6.6).

Use, e.g., a quadratic polynomial for the velocity in the method of manufactured solutions. The expected error is $\mathcal{O}(\Delta t^2)$ from the centered finite difference approximation and the geometric mean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the discrete equations in many problems. Show that this is true for linear drag (by adding a source term that depends on t), but not for quadratic drag because of the geometric mean approximation. Use the method of manufactured solutions to add a source term *in the discrete equations for quadratic drag* such that a linear function of t is a solution. Add a test function for checking that the linear function is reproduced to machine precision in the case of both linear and quadratic drag.

Apply the software to a case where a ball rises in water. The buoyancy force is here the driving force, but the drag will be significant and balance the other forces after a short time. A soccer ball has radius 11 cm and mass 0.43 kg. Start the motion from rest, set the density of water, ϱ, to 1000 kg/m^3, set the dynamic viscosity, μ, to 10^{-3} Pa s, and use a drag coefficient for a sphere: 0.45. Plot the velocity of the rising ball. Filename: `vertical_motion`.

Project 29: Simulate parachuting

The aim of this project is to develop a general solver for the vertical motion of a body with quadratic air drag, verify the solver, apply the solver to a skydiver in free fall, and finally apply the solver to a complete parachute jump.

All the pieces of software implemented in this project should be realized as Python functions and/or classes and collected in one module.

a) Set up the differential equation problem that governs the velocity of the motion. The parachute jumper is subject to the gravity force and a quadratic drag force. Assume constant density. Add an extra source term be used for program verification. Identify the input data to the problem.

b) Make a Python module for computing the velocity of the motion. Also equip the module with functionality for plotting the velocity.

Hint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ in time to linearize the equation of motion with quadratic drag.

Hint 2. You can either use functions or classes for implementation. If you choose functions, make a function `solver` that takes all the input data in the problem as arguments and that returns the velocity (as a mesh function) and the time mesh. In case of a class-based implementation, introduce a problem class with the physical data and a solver class with the numerical data and a `solve` method that stores the velocity and the mesh in the class.

Allow for a time-dependent area and drag coefficient in the formula for the drag force.

c) Show that a linear function of t does not fulfill the discrete equations because of the geometric mean approximation used for the quadratic drag term. Fit a source term, as in the method of manufactured solutions, such that a linear function of t is a solution of the discrete equations. Make a test function to check that this solution is reproduced to machine precision.

d) The expected error in this problem goes like Δt^2 because we use a centered finite difference approximation with error $\mathcal{O}(\Delta t^2)$ and a geometric mean approximation with error $\mathcal{O}(\Delta t^2)$. Use the method of manufactured solutions combined with computing convergence rate to verify the code. Make a test function for checking that the convergence rate is correct.

e) Compute the drag force, the gravity force, and the buoyancy force as a function of time. Create a plot with these three forces.

Hint. You can either make a function `forces(v, t, plot=None)` that returns the forces (as mesh functions) and `t`, and shows a plot on the screen and also saves the plot to a file with name stored in `plot` if `plot` is not `None`, or you can extend the solver class with computation of forces and include plotting of forces in the visualization class.

f) Compute the velocity of a skydiver in free fall before the parachute opens.

Hint. Meade and Struthers [9] provide some data relevant to skydiving. The mass of the human body and equipment can be set to 100 kg. A skydiver in spread-eagle formation has a cross-section of 0.5 m^2 in the horizontal plane. The density of air decreases varies altitude, but can be taken as constant, 1 kg/m^3, for altitudes relevant to skydiving (0-4000 m). The drag coefficient for a man in upright position can be set to 1.2. Start with a zero velocity. A free fall typically has a terminating velocity of 45 m/s. (This value can be used to tune other parameters.)

g) The next task is to simulate a parachute jumper during free fall and after the parachute opens. At time t_p, the parachute opens and the drag coefficient and the cross-sectional area change dramatically. Use the program to simulate a jump from $z = 3000$ m to the ground $z = 0$. What is the maximum acceleration, measured in units of g, experienced by the jumper?

Hint. Following Meade and Struthers [9], one can set the cross-section area perpendicular to the motion to 44 m^2 when the parachute is open. Assume that it takes 8 s to increase the area linearly from the original to the final value. The drag coefficient for an open parachute can be taken as 1.8, but tuned using the known value of the typical terminating velocity reached before landing: 5.3 m/s. One can take the drag coefficient as a piecewise constant function with an abrupt change at t_p. The parachute is typically released after $t_p = 60$ s, but larger values of t_p can be used to make plots more illustrative.
Filename: `parachuting`.

Exercise 30: Formulate vertical motion in the atmosphere

Vertical motion of a body in the atmosphere needs to take into account a varying air density if the range of altitudes is many kilometers. In this case, ϱ varies with the altitude z. The equation of motion for the body is given in Section 9.10. Let us assume quadratic drag force (otherwise the body has to be very, very small). A differential equation problem for the air density, based on the information for the one-layer atmospheric model in Section 9.8, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p, \tag{188}$$

$$\varrho = p\frac{M}{R^*T}\,. \tag{189}$$

To evaluate $p(z)$ we need the altitude z. From the principle that the velocity is the derivative of the position we have that

$$z'(t) = v(t), \tag{190}$$

where v is the velocity of the body.

Explain in detail how the governing equations can be discretized by the Forward Euler and the Crank-Nicolson methods. Discuss pros and cons of the two methods. Filename: `falling_in_variable_density`.

Exercise 31: Simulate vertical motion in the atmosphere

Implement the Forward Euler or the Crank-Nicolson scheme derived in Exercise 30. Demonstrate the effect of air density variation on a falling human, e.g., the famous fall of Felix Baumgartner. The drag coefficient can be set to 1.2. Filename: `falling_in_variable_density`.

Exercise 32: Compute $y = |x|$ by solving an ODE

Consider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

which has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, calculate by hand y_1 and y_2 from the Forward Euler, Backward Euler, Crank-Nicolson, and Leapfrog methods. Use all of the former three methods for computing the y_1 value to be used in the Leapfrog calculation of y_2. Thereafter, visualize how these schemes perform for a uniformly partitioned mesh with $N = 10$ and $N = 11$ points. Filename: `signum`.

Exercise 33: Simulate growth of a fortune with random interest rate

The goal of this exercise is to compute the value of a fortune subject to inflation and a random interest rate. Suppose that the inflation is constant at i percent per year and that the annual interest rate, p, changes randomly at each time step, starting at some value p_0 at $t = 0$. The random change is from a value p^n at $t = t_n$ to $p_n + \Delta p$ with probability 0.25 and $p_n - \Delta p$ with probability 0.25. No change occurs with probability 0.5. There is also no change if p^{n+1} exceeds 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial fortune scaled to 1, and simulate 1000 scenarios of length 20 years. Compute the mean evolution of one unit of money and the corresponding standard deviation. Plot the mean curve along with the mean plus one standard deviation and the mean minus one standard deviation. This will illustrate the uncertainty in the mean curve.

Hint 1. The following code snippet computes p^{n+1}:

```
import random

def new_interest_rate(p_n, dp=0.5):
    r = random.random()  # uniformly distr. random number in [0,1)
    if 0 <= r < 0.25:
        p_np1 = p_n + dp
    elif 0.25 <= r < 0.5:
        p_np1 = p_n - dp
    else:
        p_np1 = p_n
    return (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

Hint 2. If $u_i(t)$ is the value of the fortune in experiment number i, $i = 0, \ldots, N-1$, the mean evolution of the fortune is

$$\bar{u}(t) = \frac{1}{N}\sum_{i=0}^{N-1} u_i(t),$$

and the standard deviation is

$$s(t) = \sqrt{\frac{1}{N-1}\left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1}(u_i(t))^2\right)}.$$

Suppose $u_i(t)$ is stored in an array `u`. The mean and the standard deviation of the fortune is most efficiently computed by using two accumulation arrays, `sum_u` and `sum_u2`, and performing `sum_u += u` and `sum_u2 += u**2` after every experiment. This technique avoids storing all the $u_i(t)$ time series for computing the statistics.
Filename: `random_interest`.

Exercise 34: Simulate a population in a changing environment

We shall study a population modeled by (129) where the environment, represented by r and f, undergoes changes with time.

a) Assume that there is a sudden drop (increase) in the birth (death) rate at time $t = t_r$, because of limited nutrition or food supply:

$$r(t) = \begin{cases} \varrho, & t < t_r, \\ \varrho - A, & t \geq t_r, \end{cases}$$

This drop in population growth is compensated by a sudden net immigration at time $t_f > t_r$:

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

Start with ϱ and make $A > \varrho$. Experiment with these and other parameters to illustrate the interplay of growth and decay in such a problem.

b) Now we assume that the environmental conditions changes periodically with time so that we may take

$$r(t) = \varrho + A\sin\left(\frac{2\pi}{P}t\right).$$

That is, the combined birth and death rate oscillates around ϱ with a maximum change of $\pm A$ repeating over a period of length P in time. Set $f = 0$ and experiment with the other parameters to illustrate typical features of the solution. Filename: `population.py`.

Exercise 35: Simulate logistic growth

Solve the logistic ODE (130) using a Crank-Nicolson scheme where $(u^{n+\frac{1}{2}})^2$ is approximated by a *geometric mean*:

$$(u^{n+\frac{1}{2}})^2 \approx u^{n+1}u^n\,.$$

This trick makes the discrete equation linear in u^{n+1}. Filename: `logistic_CN`.

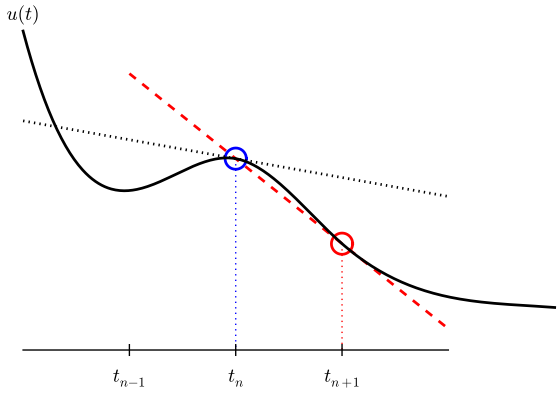
Exercise 36: Rederive the equation for continuous compound interest

The ODE model (133) was derived under the assumption that r was constant. Perform an alternative derivation without this assumption: 1) start with (131); 2) introduce a time step Δt instead of m: $\Delta t = 1/m$ if t is measured in years; 3) divide by Δt and take the limit $\Delta t \to 0$. Simulate a case where the inflation is at a constant level I percent per year and the interest rate oscillates: $r = -I/2 + r_0\sin(2\pi t)$. Compare solutions for $r_0 = I, 3I/2, 2I$. Filename: `interest_modeling`.

11 Summarizing multiple-choice questions

Exercise 37: Characterize a finite difference

Question: We can approximate the derivative at a point using two function values:



What is this type of difference called and how large is the error?

A. This is a centered difference with error proportional to Δt^2.
B. This is a forward difference with error proportional to Δt.
C. This is a Forward Euler difference with error proportional to Δt^3.
D. This is a Backward Euler finite difference with error proportional to Δt.

Answer: B.
Solution:

A: Wrong. No, a centered difference would have the two function values equally displaced to either side of the target point t_n.

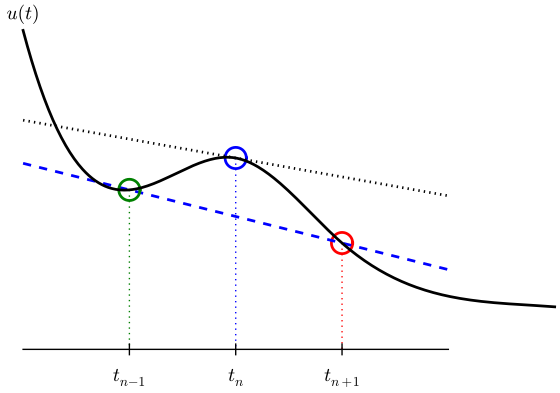
B: Right. The name is forward difference, or Forward Euler difference in the context of differential equations. In such contexts the formula is also known as Euler's method or formula.

C: Wrong. One may well call this a Forward Euler difference, but the error is not as "good" as Δt^3.

D: Wrong. Since we use the points t_{n+1} and t_n when constructing the difference, we go *forward* in time, not backward. Therefore, this is a forward difference.

Exercise 38: Characterize a finite difference

Question: We can approximate the derivative at a point using two function values:



What is this type of difference called and how large is the error?

A. This is a centered difference with error proportional to Δt^2.
B. This is a forward difference with error proportional to Δt^2.
C. This is a centered difference with error proportional to Δt^4.
D. This is a Backward Euler finite difference with error proportional to Δt.

Answer: A.
Solution:

A: Right.

B: Wrong. A forward difference makes use of the point itself, t_n, and a point *forward* in time, t_{n+1}. This is not the case here: we use points to the left and right, and the derivative is approximated in the center point. Also, a forward difference would not have an error $\mathcal{O}(\Delta t^2)$.

C: Wrong. It is centered, but the error is only $\mathcal{O}(\Delta t^2)$.

D: Wrong. Since we use the points t_{n+1} and t_n when constructing the difference at $t_{n+1/2}$, the derivative is in the *center* of the two points, and the difference is therefore a *centered* difference. A backward difference, or Backward Euler difference, would use $t_{n+1/2}$ in this case and some point $t_{n-1/2}$ *backward* in time.

Exercise 39: What is the problem with this program?

Question: We want to solve

$$u' = -au, \quad u(0) = 1,$$

by a Forward Euler scheme,

$$u^{n+1} = u^n - a\Delta t\, u^n,$$

and the following program

```
from numpy import *
u = zeros(10)
u[0] = 1
dt = 0.1
for i in range(10):
    u[i+1] = u[i] - dt*a*u[n]
```

What is the major problem with this program?

A. `from numpy import *` is not recommended; one should import explicitly the functions needed or do `import numpy as np`.

B. The program aborts with a `NameError`.

C. The program is "flat". It should be wrapped in a function `solver(U0, dt, N, a)`.

D. The scheme is unstable!

Answer: B.
Solution:

A: Wrong. True, these are recommended rules, but it is not a problem to do the "star import": it is legal and convenient.

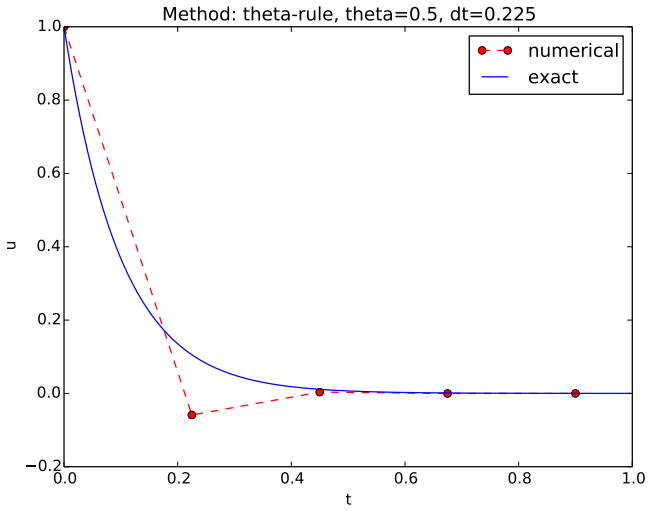
B: Right. True, `a` is not defined. This is the only reason why one cannot execute the program.

C: Wrong. That is definitely a good idea, but it is not a major problem for computing the solution of the differential equation.

D: Wrong. Actually, this can be true. Since `a` is not defined, it may happen that `dt=0.1` is a too long time step for the stability restrictions of the Forward Euler method for this differential equation. We need $\Delta t \leq 1/a$. However, the answer is wrong in the sense that this *potential* instability is not the major problem with the program - it is a bigger problem that `a` is not defined.

Exercise 40: Is the solution correct?

Question: You have solved $u' = -au$ by the Crank-Nicolson scheme and tested your program thoroughly. Suddenly you run the program with $I = 1$, $a = 10$, $T = 1$, and $\Delta t = 0.225$. You get somewhat unexpected results:



The results are unexpected because we know the exact solution should be monotone and decreasing, while this numerical solution also shows an increasing stage. What is the problem?

A. The program is not tested well enough - yet another bug is there.

B. The numerical solution method is more general than the analytical one, and this is an example where the solution can increase, but the analytical solution technique is not capable of dealing with this situation.

C. The time step is too large and cause an instability in the form of non-physical oscillations.

D. The Crank-Nicolson scheme is unconditionally stable and can be used with all time-step sizes, but the problem here is that round-off errors due to a "too large" a (10, not around 1) accumulate to the effect seen in the plot.

Answer: C.
Solution:

A: Wrong. No, this computation is in fact correct - numerically.

B: Wrong. The analytical solution $u(t) = Ie^{-at}$ covers all possible cases.

C: Right. True. One needs $\Delta t \leq 2/a = 0.2$ in this case to avoid oscillations.

D: Wrong. 1: The first part is true if unconditionally stable means that the solution is bounded and decays with time, but one may also argue that oscillations, which are non-physical, are a kind of instability, and these occur if $\Delta t > 2/a$. 2: Round-off errors are *very* small in this problem, compared to the discretization errors, and cannot cause an oscillating solution.

Exercise 41: Is this a proper test function?

Question: Suppose we have some function `compute` that we want to test. We construct a unit test and implement an associated test function (according to the rules for test functions in the nose or pytest test frameworks):

```
def test_compute(n):
    expected = (n**2)/4.5 - 1
    computed = compute(n)
    assert expected != computed
```

The question is if this test function can be used as is or if improvements must be implemented.

A. One should use special assert functions from `nose.tools`, here the choice is `nose.tools.assert_almost_equal`.

B. One cannot test `compute(n)` for only one `n` value. Many are required for good evidence that the function works.

C. The test function does not test `expected != computed` with a tolerance and the `n` parameter cannot be an argument.

D. The `assert` statement also needs a message explaining what is wrong when the test fails.

Answer: C.
Solution:

A: Wrong. It is true that an "almost equal" type of assert function is appropriate, but there is nothing in the rules that requires use of special assert functions. The requirement is that an `AssertionError` is raised if the test fails. That is done by a plain `assert` as used here.

B: Wrong. This depends on what is inside `compute`. If it has several branches depending on the value of `n`, one must test for a visit to each branch, which requires multiple `n` values, but if it is a formula (the test might indicate so), one value can be sufficient.

C: Right. The formula for `expected` indicates that this is a real number that is subject to potential round-off errors, so one should use a tolerance: `abs(expected - computed) < 1E-14`. Also, test functions should never take arguments.

D: Wrong. This is always a good idea, but not a requirement.

Exercise 42: Rewrite an expression with array arithmetics

Question: A mesh function is initialized with the code segment

```
from math import exp
N_t = 100
T = 3.0
dt = T/N_t
```

```
t = []
u = []
for i in range(len(t)):
    t.append(i*dt)
    u.append(1 - exp(-t[-1]))
```

Rewrite this code such that there is no loop and `u` is computed by array arithmetics.

A.

```
from numpy import *
from math import exp
N_t = 100
t = np.linspace(0, 3, N_t+1)
u = 1 - exp(-t)
```

B.

```
import numpy as np
N_t = 100
t = np.linspace(0, 3, N_t+1)
u = 1 - np.exp(-t)
```

C.

```
N_t = 100
dt = 3.0/N_t
t = [i*dt for i in range(N_t+1)]
from numpy import exp
u = 1 - exp(-t)
```

D.

```
import numpy
t = numpy.mesh([0, 3], 100)
u = numpy.func(numpy.exp, t)
```

Answer: B.
Solution:

A: Wrong. The `exp` function is imported from `math` (since `from math import exp` reimports the `exp` name that was imported by `from numpy import *`) and cannot be used with array argument `t`.

B: Right.

C: Wrong. The computation of `u` applies array arithmetics, but the list comprehension for `t` involves a standard (slow) Python loop.

D: Wrong. There are no functions `numpy.mesh` and `numpy.func`.

Exercise 43: What is the truncation error?

Question: What is the truncation error?

A. The difference between the exact solution and the numerical solution at a mesh point.

B. The error in the factor that takes `u[i]` to `u[i+1]`.

C. The difference between the exact solution and the numerical solution truncated to one decimal.

D. The error in the scheme when the exact solution is inserted in the scheme's difference equation.

Answer: D.
Solution:

A: Wrong. This is the global error.
B: Wrong. This is the amplification factor error or the local error.
C: Wrong. Nobody applies this error measure.
D: Right.

Exercise 44: Recognize a programming language

Question: What kind of programming language is this?

```
function integral = trapezoidal(f, a, b, n)
    %% Integrate f from a to b with n intervals
    h = (b-a)/n;
    result = 0.5*f(a) + 0.5*f(b);
    for i = 1:(n-1)
        result = result + f(a + i*h);
    end
    integral = h*result;
end
```

A. Python
B. MATLAB or Octave
C. Cython
D. FORTRAN 77

Answer: B.
Solution: **A**: Wrong. **B**: Right. **C**: Wrong. **D**: Wrong.

Exercise 45: Recognize a programming language

Question: What kind of programming language is this?

```
def trapezoidal(f, a, b, n):
    # Integrate f from a to b with n intervals
    h = (b-a)/float(n)
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        result += f(a + i*h)
    return h*result
```

A. Python
B. MATLAB or Octave
C. Cython
D. FORTRAN 77

Answer: A.
Solution: **A**: Right. **B**: Wrong. **C**: Wrong. **D**: Wrong.

Exercise 46: Recognize a programming language

Question: What kind of programming language is this?

```
      real*8 function trapezoidal(f, a, b, n)
C     Integrate f from a to b with n intervals
      real*8 h, result, f
      external f
      h = (b-a)/n
      result = 0.5*f(a) + 0.5*f(b)
      do i = 1, n-1
        result = result + f(a + i*h)
      end do
      trapezoidal = h*result
      return
      end
```

A. Python
B. MATLAB or Octave
C. Cython
D. FORTRAN 77

Answer: D.
Solution: **A**: Wrong. **B**: Wrong. **C**: Wrong. **D**: Right.

Exercise 47: Recognize a programming language

Question: What kind of programming language is this?

```
double trapezoidal(
    double (*f)(double), double a, double b, int n)
{
    double h, result;
    h = (b-a)/n;
    result = 0.5*f(a) + 0.5*f(b);
    for (i=1; i++; i <= n-1) {
        result += f(a + i*h);
    }
    return h*result;
}
```

A. C
B. C++
C. Cython
D. Octave dialect

Answer: A.
Solution: **A**: Right. **B**: Wrong. **C**: Wrong. **D**: Wrong.

Exercise 48: What is SymPy?

Question: What is SymPy?

A. A Python module for computing with symmetric matrices.
B. A Python package for doing symbolic computations (exact/analytical differentiation, integration, equation solves, etc.).
C. A Python package for numerical approximations to differentiation, integration, equation solving, etc.
D. A free, open source version of Mathematica.

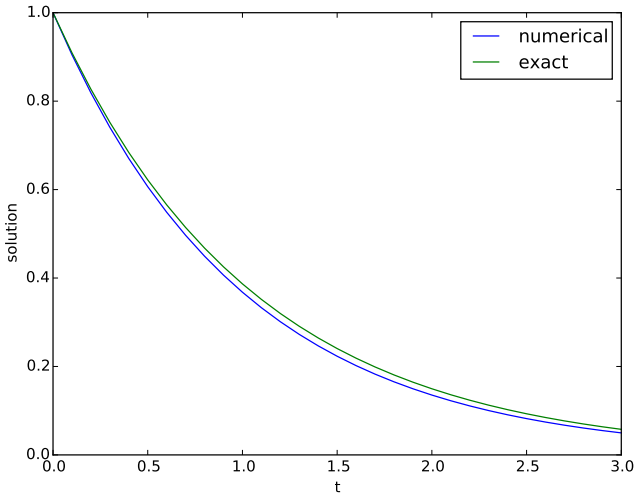
Answer: B.
Solution:

A: Wrong. SymPy can compute with symmetric matrices, so that is true, but it can do very much more.
B: Right.
C: Wrong. It's the opposite: analytical, not numerical.
D: Wrong. Could be viewed as such, but Mathematica is a much more advanced tool for symbolic computing.

Exercise 49: Testing of code

Question: What is an appropriate test for computing the solution of some ODE?

A. We compare the numerical solution with the analytical solution in a plot. The curves are quite close, showing that the computations are correct (here using 31 mesh points).



B. The maximum error between the numerical and exact solution at the mesh points is $1.886 \cdot 10^{-5}$, which is small. Therefore, the program works.

C. The scheme is $\mathcal{O}(\Delta t^2)$. With 31 points we get a maximum point-wise error of $1.243 \cdot 10^{-4}$. With 61 points (halving Δt) the corresponding error is $3.108 \cdot 10^{-5}$. This is a reduction of approximately a factor 4, which is what we expect for such a scheme when halving Δt.

D. The numerical solution and the exact solutions get closer and closer in a plot as we reduce Δt. Therefore, the program works.

Answer: C.
Solution:

A: Wrong. This comparison says nothing if the discrepancy is the unavoidable numerical error or if it also contains the effect of bugs in the program.

B: Wrong. Nobody knows if this is the numerical approximation error or if it also contains programming errors.

C: Right. This test involves checking of the convergence rate, which is a good test. The only knowledge we have of the numerical error is its rate!

D: Wrong. This test points to convergence of the method, but the error can still contain the effect of bugs. One needs to measure *how fast* the curves get closer and closer.

Exercise 50: What kind of scheme is this?

Question: Given $u' = -au + b$, where a and b are functions of time. Is the following scheme correct?

$$\frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)\frac{1}{2}(u^n + u^{n-1}) + b(t_n)\,.$$

A. Yes, it is a Crank-Nicolson type of scheme.

B. Yes, if $a(t_n)$ and $b(t_n)$ are evaluated at the previous time step, $a(t_{n-1})$ and $a(t_{n-1})$.

C. No, the coefficients a and b are not evaluated correctly.

D. No, the right-hand side should be $a(t_n)u^n + b(t_n)$.

Answer: C.
Solution:

A: Wrong. No, then a and b should have been evaluated at $t_{n-1/2}$.

B: Wrong. No, this will be wrong since the arithmetic mean on the right-hand side points sampling the ODE at $t_{n-1/2}$. Then a and b must be evaluated at this point.

C: Right. That is right: they should be evaluated at $t_{n-1/2}$.

D: Wrong. That is right if the scheme should be a Backward Euler scheme, but the arithmetic mean on the right-hand side points to a Crank-Nicolson scheme, though with wrong sampling of a and b.

Exercise 51: What kind of scheme is this?

Question: We want to solve $y' = g(x, y)$ for $y(x)$ and have the scheme

$$y^{i+1} = y^i + \Delta t\, g(x_{i+1}, y^{i+1})\,.$$

What is this scheme called?

A. The implicit midpoint scheme.

B. The Backward Euler scheme or just the backward scheme.

C. The Forward Euler scheme, Euler's method, or just the forward scheme.

D. The implicit Adams scheme of order one.

Answer: B.
Solution:

A: Wrong. There is no midpoint $i + 1/2$ involved here.

B: Right.

C: Wrong. This is true if we had $g(x_i, y^i)$.

D: Wrong.

Exercise 52: What kind of scheme is this?

Question: We want to solve $y' = g(x, y)$ for $y(x)$ and have the scheme

$$y^{i+1} = y^{i-1} + 2\Delta t\, g(x_i, y^i)\,.$$

What is this scheme called?

A. The implicit midpoint scheme.
B. The two-step backward scheme.
C. The Crank-Nicolson scheme.
D. The leapfrog scheme.

Answer: D.
Solution:

A: Wrong. It is a midpoint scheme, but it is explicit rather than implicit.
B: Wrong.
C: Wrong. No, that looks quite different and is based on a centered difference over $[x_i, x_{i+1}]$, not $x_{i-1}, x_{i+1}]$.
D: Right.

References

[1] D. Griffiths, F. David, and D. J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer, 2010.

[2] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer, 1993.

[3] G. Hairer and E. Wanner. *Solving Ordinary Differential Equations II*. Springer, 2010.

[4] J. D. Hunter, D. Dale, E. Firing, and M. Droettboom. Matplotlib documentation, 2012.

[5] H. P. Langtangen. Scaling. *http://tinyurl.com/k3sdbuv/pub/scale*.

[6] H. P. Langtangen. SciTools documentation. `http://hplgit.github.io/scitools/doc/web/index.html`.

[7] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Texts in Computational Science and Engineering. Springer, fourth edition, 2014.

[8] H. P. Langtangen and L. Wang. Odespy software package. `https://github.com/hplgit/odespy`.

[9] D. B. Meade and A. A. Struthers. Differential equations in the new millenium: the parachute problem. *International Journal of Engineering Education*, 15(6):417–424, 1999.

[10] L. Petzold and U. M. Ascher. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, volume 61. SIAM, 1998.

Index